Dr. Dobb's SOURCEBOOK

Tomorrow's Hot CPUs — Today!

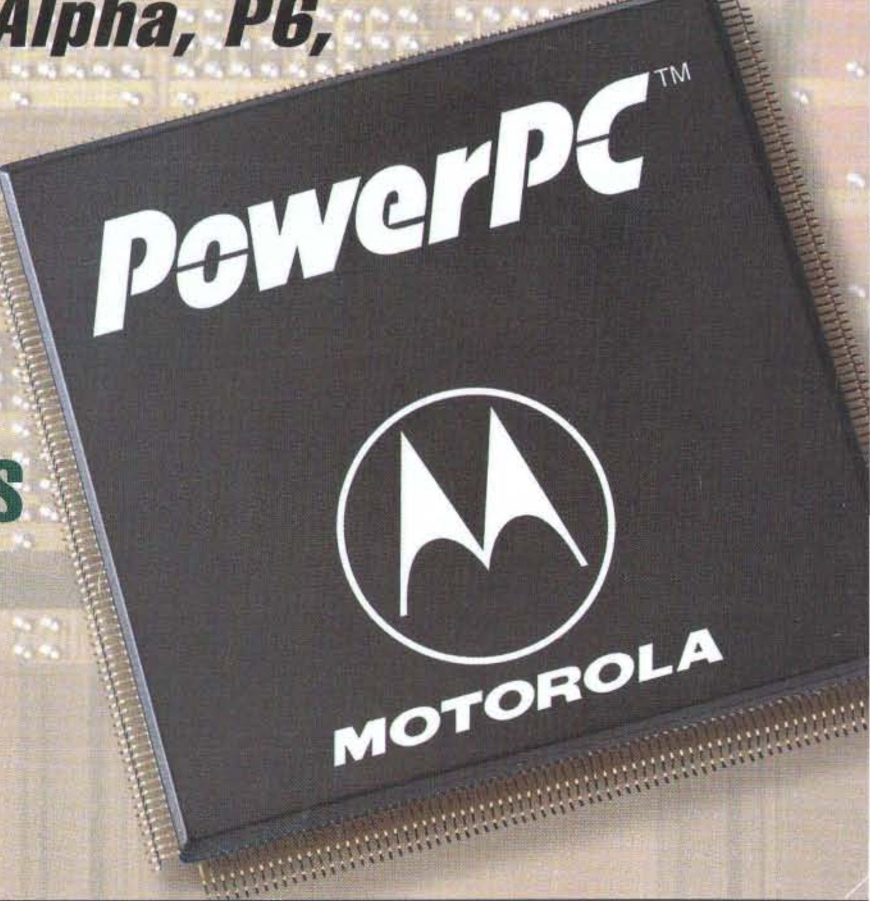
#241 SEPTEMBER/OCTOBER 1995

POWERPC PROGRAMMING

- *Optimizing for the PowerPC*
- *Porting to the PowerMac*
- *PowerPC, DEC Alpha, P6, And More!*

PATENTING YOUR SOFTWARE

ABRASH ON **3-D TRANSFORMS**



$4.95 ($5.95 CANADA)

A Miller Freeman Publication

WHERE DO YOU

WANT TO GO

TODAY?™

Good question

Microsoft.

But what about tomorrow?

Ask Windows users where they want to go today, and their answer is likely to be this: Windows 95. It is, after all, a major advance in the state of Windows computing. And it does, finally, bring some of the innovations pioneered by Apple in 1984 to the PC desktop of 1995.

While other PC manufacturers are still struggling to get CDs to load, Macintosh users can create their own multimedia, work in 3-D, surf the Internet and see what's real about virtual reality. Today.

That's great, today. But where, one has to ask, is desktop computing going tomorrow? And is moving to Windows 95 really the right way to get there?

The future of computing.

In a word, it's multimedia. Microsoft and Intel say it's the future. So do we. The difference is, we deliver that future today. To see what we mean, simply turn a Power Mac™ on. When you do, you can not only get down to work (or play) with the CD-ROM of your choice, you can also start using 3-D graphics. You can talk to your Mac.™ And have it recognize your command. You can videoconference across continents. You can even dive into virtual reality.* All at the touch of a few keys and the click of a mouse.

The power to do it.

To do all this, you need power. And the best way to get it is with a Power Mac. In recent tests, for example, the RISC-based Power Macintosh™ 9500 outperformed a 120 MHz Pentium-processor-based PC by 63% on average. When running scientific and technical apps, the performance advantage jumped to 80%. And for graphics, the Power Mac was more than twice as fast.**

Because Power Mac computers are based on the blistering fast PowerPC™ RISC chip, they have power to spare for tomorrow's advanced applications like interactive media and virtual reality.

Eleven years after it was first introduced, Macintosh is still the only personal computer in the world designed from the start as a seamless integration of hardware and software.

The easiest way to get there.

Of course, all the raw power in the world is worthless if you can't use it. That's why every new Mac includes an innovative help system that doesn't just answer your questions, but shows you what to do, where to click and what to type to get things done. And why we make it so easy to create Internet connections, install new software and set up entire new networks from scratch.

More choices than ever.

The Power Macintosh 6100/66 DOS Compatible includes both a 66 MHz 486 chip and a RISC-based PowerPC chip, making it the most compatible computer you can find.

Today, every new Macintosh® can read and write DOS and Windows disks. But our compatibility goes further than that. The Power Macintosh 6100/66 DOS Compatible, for example, runs thousands of DOS and Windows applications, in addition to thousands of programs for Macintosh. And our new Power Mac systems accept standard PCI cards.

In the future, Apple innovations will further break down the barriers between cross-platform collaboration. Distinctions between the platforms themselves will diminish. Even the boundaries between applications will blur.

All of which will add up, once again, to the most important kind of power of all. The power to be your best.®

To learn more about Macintosh power today, and tomorrow, visit us on the Internet today at http://www.apple.com.

*Some of these capabilities may require the purchase of additional software or hardware. **Don't believe us? Surf on over to http://www.apple.com to see the results of independent testing by Competitive Assessment Services, 1995. © 1995 Apple Computer, Inc. All rights reserved. Apple, the Apple logo, Macintosh and "The power to be your best" are registered trademarks of Apple Computer, Inc. Mac, Power Mac and Power Macintosh are trademarks of Apple Computer, Inc. PowerPC is a trademark of International Business Machines Corporation, used under license therefrom. "Where do you want to go today?" is a trademark of Microsoft Corporation. All Macintosh computers are designed to be accessible to individuals with disability.

CIRCLE NO. 32 ON READER SERVICE CARD

Why Settle For GIF, TIFF and JPEG When You Can Have All This...

15 Multimedia Formats

AVI	**Audio Video Interleave**	**ILBM**	**Amiga Interchange File Format**
BMP	**Microsoft Bitmap**	**JPEG**	**True Color Compressed**
CLP	**Windows Clipboard**	**PCX**	**Zsoft**
DIB	**Windows Device Independent Bitmap**	**PNG**	**The New GIF Replacement Format**
GIF ™	**Graphics Interchange Format**	**RAS**	**Sun Raster**
EPS	**Encapsulated PostScript™**	**RGB**	**Silicon Graphics**
FLC	**Autodesk™ FLIC**	**TIFF**	**Tagged Image File Format**
IFF	**Amiga Interchange File Format**	**TARGA**	**Truevision**

Add extensive multimedia support to your Windows application with just a few hours of development. The Common Format Image Libraries span the entire Windows product line including Windows 3.1, Windows 95 and all Windows NT versions! The libraries use a consistent interface across all platforms which eases cross-platform development. By simplifying coding efforts these libraries ***will substantially reduce*** your development time. The libraries were developed for intensive use in commercial applications, contain extensive error checking and have been field proven over several years.

Pricing for the Libraries:

Windows 3.1	***$750.00***
Windows95	***$1000.00***
Intel NT	***$1000.00***
PowerPC NT	***$2000.00***
MIPS NT	***$2000.00***
ALPHA NT	***$3500.00***

Download our API, a sample library, and documentation from our Web Page located at http://cblx.unh.edu/ncs.htm

Try these royalty-free Libraries Risk-Free with our 30 Day Money-Back Guarantee!

Quality problems with AccuSoft™ libraries? Compatibility problems with LEAD™ libraries? Take $300.00 OFF with our Risk-Free Competitive Upgrade!

PC MAGAZINE EDITORS' CHOICE April 13, 1993

Conversion Artist

DV AWARD OF EXCELLENCE

PhotoMorph

MULTIMEDIA WORLD ALL-STAR

PhotoMorph

PC MAGAZINE BEST OF 1993 Jan 11, 1994

PhotoMorph

CIRCLE NO. 25 ON READER SERVICE CARD

North Coast Software, Inc.

P.O. Box 459 • Barrington, NH 03825

Tel. (603) 664-7871 • Fax (603) 664-7872 • Email: 4386449@mcimail.com

CONTENTS

September/October 1995
Volume 20, Issue 16
PowerPC Sourcebook

PROGRAMMER'S SERVICES

As a service to our readers, all source code is available on a single disk and online. To order the disk, send $14.95 (California residents add sales tax) to *Dr. Dobb's Journal,* 411 Borel Ave., San Mateo, CA 94402, call 415-655-4100 x5701, or use your credit card to order by fax, 415-358-9749. Specify issue number and disk format. Code is also available through the DDJ Forum on CompuServe (type GO DDJ), via anonymous FTP from site ftp.mv.com(192.80.84.3) in the /pub/ddj directory, on the World Wide Web at http://WWW.ddj.com, and through DDJ Online, a free service accessible via direct dial at 415-358-8857 (1200/2400/9600 baud, 8-N-1).

EDITORIAL

Rebel Alliance

Believing they needed a competitive edge to combat a common foe, a small band of large companies, including IBM and Apple, formed an alliance more than four years ago. The mission of the alliance was to create a new computer capable of running each vendor's software, while providing advantages over that of the "Wintel" (Windows running on Intel-based platforms) empire. The alliance decided that, if it could base its approach on a RISC architecture that offered price and performance advantages over the empire, it might have a fighting chance. And while Intel was protecting its chip's architecture through patents, the alliance agreed to create a common instruction set architecture (ISA), allowing other chip makers to create processors that will run the same code.

Four years later, both Apple and IBM have launched hardware platforms based on the Motorola/IBM PowerPC chip. Now, several operating systems are either in development or already on the market, including AIX, Solaris, Linux, OS/2, Windows NT, and Apple's Copland. Additionally, developer tools are beginning to show up in numbers. The big question now is whether corporate developers will embrace the new technology.

Developers must consider a dizzying array of options. First, there was the original PowerPC 601 microprocessor, manufactured by IBM but sold by IBM and Motorola. The 601 was designed to bridge the gap between PowerPC and the POWER chip used in IBM's RS/6000 workstations. Thus, it uses the older POWER instruction set, which has since been eliminated from the specification. The MPC603 and MPC603e included power-management functionality for notebook computers. The 64-bit MPC620 chip is still in development. However, features and options such as instruction and data caches all vary with the different implementations. For instance, the MPC602 (which is targeted at consumer electronics and embedded applications) has dual 4-KB instruction and data caches. The MPC604, on the other hand, contains dual 16-KB caches, while the MPC620 will have separate 32-KB instruction and data caches.

The alliance had originally intended to deliver on its performance promises by now. Although Motorola contends that its MPC604 chip outperforms the Pentium, the estimated 15–30 percent improvement in performance still falls short of the 2:1 increase originally promised. Moreover, Intel's upcoming P6 microprocessor is expected to show performance comparable to that of the MPC604 chip.

IBM finally began shipping its PowerPC-based machines in June of this year. However, the entry-level models running Windows NT cost consumers some $3700, well above the $2500 for similar Intel-based machines running the same operating system. Further, IBM announced nearly a year ago that it was delaying the launch of its PowerPC computers so that it could make ready its OS/2 for PowerPC. Apparently, the port of the operating system is taking longer than expected, and the company could wait no longer to deliver its Power Series computers.

Meanwhile, the first Linux kernel for PowerPC is up and running on Motorola's PowerPC VME 1603. But when asked about a similar port to PowerMac, project coordinator Joseph Brothers indicates that Apple cannot come up with the necessary programming specifications for the PowerMac's NuBus, nor can it provide necessary information on devices, memory maps, or interrupt hardware. Motorola has tried for more than a year to obtain the necessary specifications from Apple. (Incidentally, the Linux kernel is available via anonymous ftp from liber.stanford.edu/pub/linuxppc.)

Despite the short-term glitches in cost, performance, and roll-outs, the PowerPC is impressive, and most major operating systems will likely run on PowerPC-based platforms in the future.

Still, at this stage, the fate of the PowerPC alliance is in the hands of developers such as yourself. May the source be with you.

Michael Floyd

Michael Floyd
Executive Editor

Dr. Dobb's SOURCEBOOK *SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER*

VICE PRESIDENT-PUBLISHER *Peter Hutchinson*

EDITORIAL
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Tami Zemel*
EXECUTIVE EDITOR *Michael Floyd*
SENIOR TECHNICAL EDITOR *Ray Valdés*
SENIOR PRODUCTION EDITOR *Monica E. Berg*
ASSOCIATE MANAGING EDITOR *John Dorsey*
EDITORIAL ASSISTANT *Kevin M. Carlson*
EDITORIAL INTERN *Johnny Hsu*
ART DIRECTOR *Bruce Gardner*
CONTRIBUTING EDITORS *Al Stevens, Tom Genereaux, Andrew Schulman, David Betz, Bruce Schneier*
EDITOR-AT-LARGE *Michael Swaine*
PRODUCTION MANAGER *Amy Shulman*
COVER PHOTOGRAPHER *Michael Carr*

CIRCULATION
DIRECTOR OF CIRCULATION *Jerry Okabe*
CIRCULATION MANAGER *Michael Poplardo*

MARKETING/ADVERTISING
SALES DIRECTOR *Charlie Shively*
ADVERTISING COORDINATOR *Sara D. Wood*
BUSINESS MANAGER *Lisa Krejci*
MARKETING *Valerie Dow, Mike Kuehl, Christopher Clarke*
ACCOUNT MANAGERS see page 38

MILLER FREEMAN INC.
CHAIRMAN/CEO *Marshall W. Freeman*
EXECUTIVE VP/COO *Thomas L. Kemp*
SENIOR VICE PRESIDENT/CFO *Warren "Andy" Ambrose*
SENIOR VICE PRESIDENTS *David Nussbaum, H. Verne Packer, Donald A. Pazour, Wini D. Ragus*
VICE PRESIDENT/SOFTWARE *Regina Starr Ridley*
VICE PRESIDENT/PRODUCTION *Andrew A. Mickus*
VICE PRESIDENT/CIRCULATION *Jerry Okabe*

Miller Freeman
A United News & Media publication

***DR. DOBB'S JOURNAL* (ISSN 1077-9019)** is published monthly by Miller Freeman Inc., 600 Harrison Street, San Francisco, CA 94107; 415-905-2200. Second-class postage paid at San Francisco and at additional entry points.

ARTICLE SUBMISSIONS: Send manuscripts and disk (with article, listings, and letter to the editor) to *DDJ* Submissions, 411 Borel Ave., San Mateo, CA 94402-3522.

***DDJ* ON COMPUSERVE:** Type GO DDJ.

***DDJ* ON THE INTERNET:** editors@ddj.com or http://WWW.ddj.com.

SUBSCRIPTION: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid in U.S. funds drawn on a U.S. bank. Canada and Mexico: $45.00 per year. All other foreign: $70.00 per year.

FOR SUBSCRIPTION QUESTIONS, change of address, and orders, call toll-free **800-456-1215** (U.S. and Canada). For all other countries, call **303-678-0439** or fax **303-661-1994**. On CompuServe, the customer-service number is 71572,341. Or write, *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188.

POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **GST (Canada) # R124771239**.

FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 30 Montgomery St., Jersey City, NJ 07302; 212-332-7100.

Make your Macintosh an Internet server and a UNIX workstation.

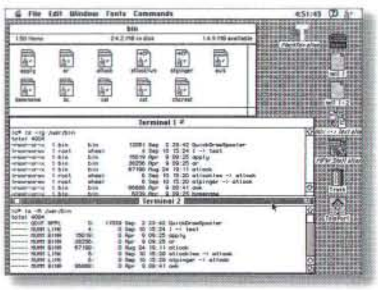
Mach^Ten is a Berkeley BSD UNIX that runs on the Classic to the Power Macintosh, including PowerBooks and Duos! So in addition to all of the easy-to-use applications that make Macintosh one of the most personable computers around, you get a MACH-based UNIX with pre-emptive multitasking.

Mach^Ten's strength lies in the way it extends the Macintosh Operating System with UNIX networking and software development tools. The Macintosh/UNIX integration is so strong that you can even use Mac programs & utilities on UNIX data, and UNIX programs & utilities on Mac files.

And Mach^Ten's full internet protocol support lets you use your Mac as a domain name server, IP router, POP mail server, or Web server.

Any NFS server can be used to store Macintosh files. Users can access them by double clicking as on the local disk.

The UNIX software development system includes the GNU C and C++ compilers and libraries to let you create new applications or port existing ones. The Motif toolkit and suite of X clients and X client libraries make developing distributed applications a breeze.


Files and directories can be viewed on disk using the Macintosh Finder or the more versatile UNIX commands.

Access network resources using the power of TCP/IP. Telnet, ftp, rlogin, or Xterm connections are possible using Mach^Ten.

And Tenon's high performance X Server lets you use your Macintosh or Power Macintosh as an X terminal.

Join the many satisfied users of proven, reliable Mach^Ten UNIX, and start turning all of your Macs into open systems today!

For more information, or to order

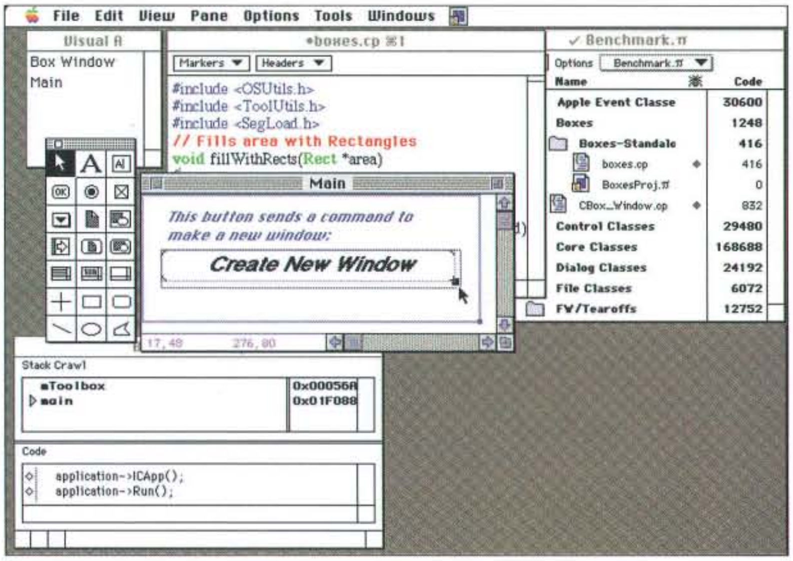
Call 1-800-6-MACH-10.

Internet: info@tenon.com
http: //www.tenon.com

CIRCLE NO. 30 ON READER SERVICE CARD

TENON INTERSYSTEMS *New Dimensions in Personal Workstation Technology*

Tenon Intersystems 1123 Chapala Street Santa Barbara, CA 93101 Tel: 805-963-6983 Fax: 805-962-8202



```
#include <OSUtils.h>
#include <ToolUtils.h>
#include <SegLoad.h>
// Fills area with Rectangles
void fillWithRects(Rect *area)
```

This button sends a command to make a new window:

Name	Code
Apple Event Classe	30600
Boxes	1248
Boxes-Standalo	416
boxes.cp	416
BoxesProj.π	0
CBox_Window.cp	832
Control Classes	29480
Core Classes	168688
Dialog Classes	24192
File Classes	6072
FW/Tearoffs	12752

```
▪Toolbox    0x00056A
▷ main      0x01F088
```

```
application->ICApp();
application->Run();
```

Boost your productivity with our Visual Architect, Advanced Editor, Debugger and Project Manager.

THE INDUSTRY STANDARD JUST MOVED TO A HIGHER POWER. SYMANTEC C++ 8.0 FOR POWER MAC.

It's the highly-regarded industry standard, used by more developers than any other Macintosh development system. And now it's been totally re-engineered for Power Mac.

Introducing Symantec C++™ 8.0 for native Power Mac.

Not since the original THINK C™ has anything so dramatically boosted your productivity. There's a Visual Architect™ to generate GUI code instantly. An advanced Project Manager to handle the largest and most complex jobs. Plus native Power Mac tools for radically improved performance.

DRAW ON THE INDUSTRY STANDARD.

With Symantec C++ 8.0, you simply draw the user interface – including windows, dialogs, controls, icons and menus. Then the built-in Visual Architect generates the code with the click of a mouse. Now you can spend more time on what really sets your application apart – its functionality.

A HIGHER STANDARD FOR SPEED.

The new high-performance compiler is dramatically faster than the previous version, so you can become more productive than ever.

And for even more power, we've added an Advanced Project Manager. It gives you drag and drop so you can easily add files, Named Option Sets for changing complex sets of options fast, plus support for even the largest applications.

There's also a new editor and a browser for modifying and navigating source files, a new debugger, an incremental linker, THINK Class Library™ 2.0 and much more.

NEW! FAST C AND C++ COMPILERS
for both Mac and Power Mac

NEW! VISUAL ARCHITECT
for the Power Mac

NEW! THINK CLASS LIBRARY
2.0 *for Power Mac*

NEW! POPUP MENUS
take you right to declarations & headers

NEW! DEBUGGER
for nested projects and shared libraries

NEW! PROJECT MANAGER
for multiple targets and hierarchical support

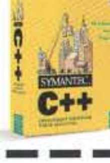
NEW! SPLIT PANE EDITOR
for syntax highlighting and auto formatting.

A DOUBLE STANDARD FOR 68K AND POWER MAC.

Symantec C++ 8.0 is the first native Mac/Power Mac development system to support C++ templates, nested classes and multiple inheritance, as well as ANSI C. Plus 8.0 includes version 7.0 for 68K support.

This double standard gives you everything you need for both 68K and Power Mac development.

To order at a special upgrade price of $149.95, call 1-800-628-4777 Ext. 9A25. Be sure to ask about the Symantec Developer's Advantage Program for premium support and regular updates. Or visit your local software store.



SYMANTEC.®

Price good in U. S. only. For more information in Canada, call 1-800-667-8661, ext. 5513. In Australia, call 2-879-6577. In Europe, call 31-71-353111. Symantec C++, Visual Architect, THINK Class Library are trademarks of Symantec Corporation. All other trademarks are the property of their respective holders. © 1995 Symantec Corporation. All rights reserved.

CIRCLE NO. 17 ON READER SERVICE CARD

Porting to the PowerMac

A tale of two operating systems

Paul Kaplan

Apple's new generation of Macs is based on the Motorola PowerPC RISC processor. The PowerMac offers extremely high performance for applications that are compiled and linked for it. However, to preserve the investment users may have in existing software, the PowerMac supports legacy 68K applications. This support is accomplished through software emulation of the 68K instruction set and operating-system support for the 68K run-time model. In addition, new and old code (as well as run-time architectures) can be mixed within an application. Apple developed the PowerMac OS this way— some of System 7.5 is still 68K code.

In this article, I'll describe the main similarities and differences between the old and new OSs and the process of porting Macintosh 68K applications to the PowerMac. I'll also present an application that illustrates using code resources to mix new and old code within an application.

68K versus PowerMac

On a PowerMac, two operating systems coexist in parallel — the original 68K system and the new PowerMac system. They run on top of a "nanokernel," which provides the lowest-level services such as memory management and interrupt handling. The magic of coexisting 68K and PowerPC software is worked by the Mixed Mode Manager.

Paul is a staff engineer with Symantec's Development Tools Group and works on Macintosh and Windows development tools. He can be contacted at pkaplan@symantec.com.

When an application is launched, the PowerMac OS looks for the special Code Fragment Resource, type *cfrg*, which specifies a PowerMac application. If a valid *cfrg* resource exists, the application is handed to the Code Fragment Manager (CFM). This subsystem manages the loading and execution of applications and shared libraries. In addition to handling the default load format, the CFM allows the use of custom loaders. A 68K application has no *cfrg* resource and is therefore handed to the 68K Segment Manager.

After an application has been launched as either 68K or PowerMac, it can switch modes while running. To switch modes and run unmodified, 68K applications call the Mixed Mode Manager implicitly; PowerMac applications can call it implicitly or explicitly.

In order to run 68K applications, the PowerMac OS has retained a number of components of the 68K OS. In fact, the PowerMac toolbox calls are a superset of the 68K system. The file system is the same, so "well-behaved" applications can be ported with little more than recompiling and linking— the development system will take care of run-time details. The System 7 MacOS, on the other hand, retains a single address space for all running applications, and the multitasking model is still cooperative and non-preemptive. Future releases of the MacOS, beginning with System 8 (code named "Copland") will provide multiple, virtual address space, preemptive multitasking, memory-mapped I/O, and object-oriented user-interface components.

The run-time model for PowerMac applications is completely new. Now, only one code and one data segment are required, and the segment manager is no longer used.

The code segment has no relocations, which makes it sharable, and all the relocations are in the data segment. Each application has a Table Of Contents (TOC) that serves the same function as the 68K "A5 world" and greatly simplifies access to global data. The TOC is created by your development system and is transparent to C or C++ code. Also, the new OS supports, and depends heavily on, shared libraries. In fact, the PowerMac toolbox is a shared library. Finally, the application file format has been completely reorganized.

Porting Your Application

Porting to the PowerPC can be as simple as recompiling if your source code meets the requirements listed in the next few paragraphs. For example, Symantec C++ 8.0 automatically converts your existing 7.0 (68K) project; recompile and link it, and it's ready to go. On the other hand, legacy applications that take shortcuts to system features will need some porting work.

The first step in porting any application is to ensure that your code runs under 68K System 7. Such an application should use only "32-bit clean" addresses. Older Mac applications sometimes used the high byte of an address for purposes other than the address. PowerPC addresses use all 32 bits. In compiling your code, use ANSI C or C++, which will force stronger type checking and function prototypes. Also, compile with Apple's Universal Headers, which are shipped with your development system. Universal Headers are appropriate for both 68K and PowerPC applications and will make your code portable between them. In addition, either rewrite inline assembly in C, or place the inline code in separate assembler files. If you insist on keeping the 68K code, it should be isolated in a separate code resource.

Don't make assumptions about registers, especially passing parameters, as they are all different. And try to use data types with 4-byte alignment. Although the PowerPC processor allows alignment anywhere, 4-byte alignment produces more-efficient code. However, if you're writing structures to a file, using 4-byte alignment can waste disk space.

Beyond these steps, you can use *#pragmas* to force 68K alignment where it is necessary for toolbox routines. Check that the alignment is correct when reading data from an existing disk file. Also, use *int* and *long* data types. On the PowerPC, *int* and *long* are 32 bits, and *short* is 16 bits. The 32-bit integer is the most efficient data type.

Use the *double* data type for floating-point variables. The PowerPC FPU supports only the IEEE 4-byte (*float*) and 8-byte (*double*) floating-point formats. *Double* is more efficient. The 10- and 12-byte *doubles* used on 68K are not supported by the processor. *Long doubles* are supported with two *doubles*. (Note that the Symantec compiler does not support *long doubles*.) Check all *#pragmas* and dependencies on *#defines* to ensure they still have meaning in the new environment. Do not put data in code. This would affect pipelined-instruction performance. And if you have Pascal code, convert it to C either by hand or with the MPW p2c Pascal-to-C converter (available on Apple's ETO #17 CD-ROM).

When porting the system interface portion of your application, you should generally use system calls instead of accessing the hardware directly. In addition, convert callbacks to universal procedure pointers. These are available in the Universal Headers. If you're passing a callback procedure's address to the operating system, you must create a *UniversalProcPtr* with the *NewRoutineDescriptor* function (the actual data structure that describes the function is called a "routine descriptor"). You need to use *UniversalProcPtrs* because the OS makes no assumption about the callback's architecture. Strictly speaking, routine descriptors are not required for 68K builds (they are compiled into addresses), but using them will make your code completely portable between the two environments.

Another thing to watch for is direct access of low memory. Don't do it! Rather, use the *LMSetxxx* and *LMGetxxx* calls in LowMem.h. Finally, don't explicitly use the 68K run-time model. The 68K run-time-specific calls are not supported. For example, a call to the Segment Manager would return with no action.

Linking Your PowerMac Application

Your linker will create a "fragment," which is the atomic load unit and contains code and static data. Fragments are managed by the CFM. Most PowerMac applications and shared libraries use the Preferred Executable Format (PEF) to house fragments. PEF specifies the file header, segments for code and data, import- and export-symbol tables, and relocations. Normally, the application resides in the data fork of its file, although fragments can be resources as well. The linker in your development system will handle the details of fragments and the PEF.

Your linker should support the xcoff format, which is an extension of the coff format found in UNIX. This is important because the only stub libraries Apple supplies to link to the toolbox and shared-library extensions such as the Drag Manager are in xcoff format. The stubs are supplied with your development system. The xcoff format can also be used to link third-party static libraries and object modules from a single translation unit. Normally, the Symantec development environment skips the step of writing object files; the compiler passes them directly to the linker in memory.

Dividing applications into shared libraries will make your code reusable and smaller by eliminating redundantly loaded code. Your development environment will help you create and manage shared libraries.

Under the Application Hood

As mentioned, the run-time model of an application running on the PowerMac OS is quite different from that of the 68K. The PowerMac run-time model has one code and one data segment, which are normally loaded in memory. The code segment is read only, which makes it suitable to run in ROM, but unsuitable to store writable data. Code and data elements may be exported from the fragment, which means their symbols are made public and may be linked dynamically. With the Symantec environment, symbols are exported with a *#pragma*.

Within the data segment resides the TOC, which is like a personal address book. It provides linkage to symbols inside and outside the fragment. The TOC has linkage to imported routines, imported data, global variables, and the pool (or pools) of static variables. When loading the application and its shared libraries, CFM resolves imported symbols and fills in the appropriate TOC entries. The TOC is 64 KB, so there is a maximum of 16K TOC entries. Your development system will warn you of a TOC overflow.

Applications should have a *main()* entry point and may additionally have user-initialization and termination routines. CFM will call the *main()* entry point of an application after it is loaded. CFM may also call an initialization routine as part of loading the fragment, and it may call a termination routine when it unloads the fragment. Your development system will help you define these entry points.

Shared-Library Details

Although common in UNIX, shared libraries are probably best known as DLLs in Microsoft Windows. Originally, shared libraries were available as an add-on to older MacOS versions with Apple Shared Library Manager (ASLM), but they are now a standard feature and are in common use on the PowerMac. Shared libraries are similar to applications. The main differences are that the file type for a shared library is *shlb*, not APPL, and that there is no *main()* entry point. Initialization and termination routines are allowed.

When the PowerMac system starts up, its shared libraries are registered with CFM and made available to all calling applications. Other shared libraries can be loaded and called at application startup if specified in the PEF file, or loaded on request

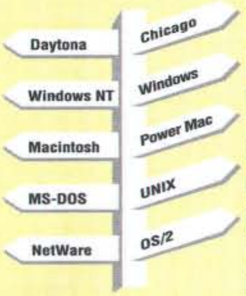
Power™

Introducing the Power Macintosh® 9500.

Visit us on the Internet at http://www.apple.com.

CIRCLE NO. 33 ON READER SERVICE CARD

WHETHER YOU'RE HEADED TO CHICAGO, DAYTONA, OR POINTS UNKNOWN, BASICSCRIPT HELPS



YOU ARRIVE IN STYLE.

Summit BasicScript Toolkit makes it easy and economical to add an award-winning scripting language to your application.

- Easy-to-use APIs make it simple to integrate BasicScript into your application using C, C++, Visual Basic, or other tools.
- Compatible with the syntax of Microsoft Visual Basic and Visual Basic for Applications (VBA).
- Controls OLE Automation objects in Windows, Windows NT, Windows 95, and Macintosh applications.
- Extensibility architecture allows you to add your own keywords to BasicScript.

CALL TODAY FOR YOUR FREE EVALUATION COPY, 315 445.9000

- Available for Windows 3.1, Windows 95 ("Chicago"), Windows NT 3.5 ("Daytona"), MS-DOS, SunOS, Solaris 2.x, HP-UX, IRIX, AIX, SCO UNIX, Ultrix, Digital UNIX, UnixWare, Macintosh, Power Macintosh, NetWare, OS/2, and OpenVMS.
- Runtime can be redistributed by your customer without royalties.
- Free technical support, comprehensive documentation, and numerous code samples keep your engineering costs to a minimum.
- Licensed by Symantec, Delrina, Intel, and other leading companies. Named *PC Magazine* Editors' Choice among cross-application macro languages.
- Flexible licensing terms mean you can afford the finest scripting language on the market today.

BASICSCRIPT

Summit Software Company
Fax: 315 445.9567
CompuServe: 71211,3504
Internet: info@summsoft.com
WWW: http://www.summsoft.com

CIRCLE NO. 23 ON READER SERVICE CARD

(continued from page 8)

by the application. Shared libraries can be loaded automatically by specifying them as import libraries to your development environment. Your linker will resolve external symbols to a shared library as though they were part of a statically linked library. However, the linker knows they've been imported and will put them in the import list for the appropriate library. As CFM loads your application, it will also attempt to load shared libraries specified by the application.

Shared libraries can also be loaded explicitly with the toolbox call *GetDiskFragment*. In this case, imports should be specified as "weak" so the linker won't be unhappy with the unresolved references. If you load a shared library explicitly, your code should be able to handle a failed load or an unresolved import (which will have a null address at run time). Shared libraries also have version capabilities. CFM checks the version number of a shared library against the version number required by the application and fails on load if it is not compatible. Version numbers can be specified by your development system.

Code Resource Examples

MacOS System 7 code resources such as CDEF and MDEF do not need to be immediately ported to PowerPC. However, there are performance penalties for mixed-mode switching and for running 68K-emulated code. If the performance of a code resource is critical, you should convert that resource to a native, or "accelerated," resource.

To illustrate the process of gradually porting to the PowerMac, I've included a sample project and the required modifications. Listings One and Two show the project source files from a 68K program that calls a 68K resource; listings begin on page 12. This project is a simple application that creates a window, has a standard event loop, and calls the *main()* routine in the code resource to handle the *Update* Event. The examples don't use any C++ features, although they were compiled with the Symantec C++ compiler. The *InitToolboxStuff()* and *MouseDownProc()* routines are standard Mac idioms and aren't shown. Also, the error checking that would be in commercial-grade code is omitted.

The first modification is the same project ported to the PowerMac. Note that the code-resource routine is still 68K and therefore unchanged. The main-project routine (see Listing Three) requires a few changes to call the code resource through the Mixed Mode Manager. Note the use of the Toolbox routine *CallUniversalProc()*, which has a *varargs* parameter list, and the two required parameters, *ProcInfoType* and *UniversalProcPtr*. *ProcInfoType* has been initialized to describe the interface of the routine so that *CallUniversalProc()* will use the parameters correctly.

The second modification illustrates the changes required to port the resource to the PowerMac. This time, the main project routine has not changed because it was ported in the first modification. Listing Four illustrates the accelerated resource code. There are new calls to *__cplusrsrcinit()* and *__cplusterm()*; the calls to *RememberA0()*, *SetupA4()*, and *RestoreA4()* have been deleted.

Normal, nonresource applications always follow the *main(argc, argv)* convention. The standard run-time library contains hidden code to set up any arguments to *main()*, and initialize static constructors and destructors. Code resources, by tradition, do not necessarily conform to an entry-point standard.

The Symantec solution for code resources in C++ requires explicit calls to the run-time routines *__cplusrsrcinit()* and *__cplusterm()* within the *main()* routine of the code resource. The run-time routines call any static constructors and destructors, and make the QuickDraw globals available to the code resource. Code resources also require routine descriptors, which play a similar role to the *ProcInfoType* parameter used in *CallUniversalProc*.

Another feature of the PowerMac is support for a "fat application"— a single Mac app that contains a 68K version in the resource fork and a PowerMac version in the data fork. Many of the resources, such as menus and icons, can be directly shared. With a little work, code resources can be shared as well. A fat application is backward compatible with 68K, System-7 machines. Although they take up more disk space, fat applications neatly solve the packaging problem for some vendors.

Conclusion

Porting standard applications from 68K to PowerPC is relatively simple. The tools you have to work with—the Mixed Mode Manager, CFM, and your development system—will allow you to gradually port your application, develop an application that will run on both the PowerMac and 68K systems, and create an application exclusively for the PowerMac.

Acknowledgments

I'd like to thank Jim Laskey, Yuen Li, John Micco, and Susan Rona, all from Symantec, for their help with this article.

DDJ

(Listings begin on page 12.)

Take a Power Trip to a New Realm of Developing

Powerful Tools to Take You Where You Want to Go

You've heard about the speed of PowerPC™ and you're thinking about the cool applications and content this amazing new chip will allow you to create. Read on to learn more about the tools that will take your development where you want to go, fast.

Prototyping HyperCard® 2.3 provides a robust prototyping and development environment for professional developers. It is also an easy-to-use tool for building a wide range of custom software applications for Macintosh®, from training simulations to information kiosks. And now it's accelerated for Power Macintosh®.

PowerPC and 68K Compilers Metrowerks CodeWarrior Gold is the industry's first native development environment for both the PowerPC and 68K-based Macintosh environments. You can easily target the entire Macintosh installed base with code created using one development environment running at Power Macintosh speeds.

Cross-Platform Multimedia Create innovative, interactive multimedia titles which will play on both Mac OS and Windows systems. Give users the ability to cruise up, down and around objects. They can even zoom in and pick them up. No wonder QuickTime™VR Authoring Tools Suite v.1.0 has won so many awards, including MacUser Magazine's "Eddy" for Breakthrough Technology of the Year.

There's a wide variety of development tools available through the Apple Developer Tools Catalog. Check it out. Developing on the Macintosh platform has never been easier or faster. And now with cross-platform flexibility, it's smarter than ever.

QuickTime VR Authoring Tools Suite v.1.0

Develop cross-platform multimedia products with QuickTime VR content

$495

This suite includes complete documentation for planning, designing, photographing and capturing content. QuickTime VR uses breakthrough compression/decompression techniques which produce small file sizes. And there's no special hardware required.

How to Order

To order call:

1 800 282-2732

And ask for a free copy of the Apple Developer Tools Catalog.
Visa, MasterCard and AMEX accepted.

Metrowerks CodeWarrior Gold

A complete development solution for C/C++ and Pascal programming.

$399

Runs native and cross-compiles on 68K or PowerPC platforms, ensuring that as you go forward on Power Macintosh you also conserve your investment in your installed code base on 68K Macintosh and your code base remains backward-compatible. Lets you compile, link, and debug 68K software at full native speed on a PowerMac™, and build PowerPC versions of your software on a 68K Mac.

HyperCard 2.3

A robust prototyping and development environment accelerated for Power Macintosh

$99

Comes with powerful scripting tools including AppleScript® Runtime built-in, a modeless script editor, hypertext support, debugging tools, new 24-bit color painting tools, Automated Button Tasks, and many other features. Stacks can be saved as stand alone, double-clickable applications you may distribute without software royalty fees.

Listing One

```
// Macintosh application to create a very simple window and do basic
// event handling. Paul Kaplan - Symantec Corporation

#include "InitToolboxStuff.h"
#include "MouseDownProc.h"
#include "UpdateWinProc.h"
#define WIN_RESID 128
#define CODE_RESID 128

void main()
    {
    static WindowPtr theWindow, foundWindow;
    static EventRecord theEvent;
    Handle UpdateWinProcHandle;

    InitToolboxStuff();

    // Setup Window and mouse tracking region
    theWindow = GetNewWindow(WIN_RESID, nil, (GrafPtr)-1);
    RgnHandle mouseRgn = NewRgn();

    // Get code resource and lock its handle
    UpdateWinProcHandle = GetResource('CODE', CODE_RESID);
    HLock(UpdateWinProcHandle);

    Boolean more2do = TRUE;
    while (more2do)     // Standard event loop processing
        {
        if (WaitNextEvent(everyEvent, &theEvent, 0xffffffff, mouseRgn))
            {
            switch(theEvent.what)
                {
                case updateEvt: // Call the code resource !!

                (*(UpdateWinProcPtr)(*UpdateWinProcHandle))(theWindow);
                break;
                case mouseDown: // Standard Mac Toolbox handling of Mouse Down
                    more2do = MouseDownProc(&theEvent, &foundWindow);
                default:
                    break;
                }
            }
        }

    // Free all allocated memory
    HUnlock(UpdateWinProcHandle);
    ReleaseResource(UpdateWinProcHandle);
    DisposeRgn(mouseRgn);
    DisposeWindow(theWindow);
}
```

Listing Two

```
// Code resource procedure to draw text in a window

#include <SetUpA4.h>
#define HORIZ 65
#define VERT 95

void main(WindowPtr myWin)
    {
    static char msg[] = "68K Code Resource";
    GrafPtr savedPort;

    RememberA0();                  // Save value of A0 for next macro
    SetUpA4();                     // Set up A4 for resource globals

    GetPort(&savedPort);           // Save current GrafPort
    SetPort(myWin);                // Make mine the current GrafPort

    BeginUpdate(myWin);

    MoveTo(HORIZ, VERT);                // Move cursor to position
    DrawText(msg, 0, sizeof(msg));  // Draw the string

    EndUpdate(myWin);

    SetPort(savedPort);            // restore current GrafPort
    RestoreA4();                   // Restore A4
    }
```

Listing Three

```
// Macintosh application to create a simple window and do basic event handling.
// Paul Kaplan - Symantec Corporation

#include "InitToolboxStuff.h"
#include "MouseDownProc.h"
#include "UpdateWinProc.h"
#define WIN_RESID 128
#define CODE_RESID 128

void main()
    {
    static WindowPtr theWindow, foundWindow;
    static EventRecord theEvent;
    Handle UpdateWinProcHandle;

    // Variable to hold Universal Proc Pointer
    UniversalProcPtr theUPP;
    // Proc Info Type - describes the called procedure's interface
    ProcInfoType theProcInfo = kCStackBased |
                                        STACK_ROUTINE_PARAMETER(1,kFourByteCode);
    InitToolboxStuff();

    // Setup Window and mouse tracking region
    theWindow = GetNewWindow(WIN_RESID, nil, (GrafPtr)-1);
    RgnHandle mouseRgn = NewRgn();

    // Get code resource and lock its handle
    UpdateWinProcHandle = GetResource('CODE', CODE_RESID);
    HLock(UpdateWinProcHandle);

    Boolean more2do = TRUE;
    while (more2do)     // Standard event loop processing
        {
        if (WaitNextEvent(everyEvent, &theEvent, 0xffffffff, mouseRgn))
            {
            switch(theEvent.what)
                {
                case updateEvt: // Call the code resource using
                                // CallUniversalProc instead of
                                // calling routine directly
                    theUPP = (UniversalProcPtr)*UpdateWinProcHandle;
                    // Convert dereferenced handle to UPP
                    CallUniversalProc(theUPP, theProcInfo, theWindow);
                    // Call MixedMode Manager
                    break;
                case mouseDown: // Standard Mac Toolbox handling of Mouse Down
                    more2do = MouseDownProc(&theEvent, &foundWindow);
                default:
                    break;
                }
            }
        }
    // Free all allocated memory
    HUnlock(UpdateWinProcHandle);
    ReleaseResource(UpdateWinProcHandle);
    DisposeRgn(mouseRgn);
    DisposeWindow(theWindow);
}
```

Listing Four

```
// Code resource procedure to draw text in a window

#include <new.h>

#define HORIZ 65
#define VERT 95

void main(WindowPtr myWin)
    {
    static char msg[] = "PPC Code Resource";
    static GrafPtr savedPort;

// Call any static constructors in this link unit. Also make
// QDGlobals available
    __cplusrsrcinit();

    GetPort(&savedPort);          // Save current GrafPort
    SetPort(myWin);               // Make mine the current GrafPort

    BeginUpdate(myWin);

    MoveTo(HORIZ, VERT);                // Move cursor to position
    DrawText(msg, 0, sizeof(msg));      // Draw the string

    EndUpdate(myWin);

    SetPort(savedPort);           // restore current GrafPort

    __cplusrsrcterm();            // Call any destructors in this link unit
    }
```

End Listings

Optimizing for the PowerPC

Strategies for greater performance

Michael Ross

With its great speed, low power requirements, and flexible programming model, the PowerPC represents a jump in microprocessor technology. Consequently, many developers are beginning to port their applications from the Intel architecture to the PowerPC. The PowerPC and the Pentium, however, represent two very different approaches to computer architecture, and moving applications from one platform to the other with a minimum of fuss is not always easy. You may need to change development platforms, targets, or tool vendors.

At MetaWare, we've ported our C/C++ compilers to the PowerPC. In doing so, we've learned a few tricks that I'll share with you in this article. I'll also describe some of the techniques we use to improve the code our compilers generate for the PowerPC. For the basis of my discussion I'll use the 601 model.

PowerPC Quick Tour

One interesting feature of the PowerPC is the branch processing unit (BPU); see Figure 1. The branch processor doesn't depend on either the integer or floating-point units; it works in concert with the instruction unit to keep instructions flowing. The BPU can look ahead in the instruction queue for a branch instruction and use static branch prediction on unresolved conditional branches to permit fetching instructions from the predicted target instruction stream. When prediction is correct, a branch can be performed in zero clock cycles. This feature of the PowerPC architecture is similar to the branch table of the Pentium or the branch target buffer and the reorder buffer of Intel's upcoming processor, the P6. The Pentium's 256-element Branch Table for dynamic branch prediction does not boast the same success rate as the PowerPC's BPU, and causes a 3- to 20-cycle penalty if the prediction fails. This is also true on the P6. The PowerPC BPU has more built-in capability that doesn't rely on the integer-processing unit. Mispredicted branches, on average, incur less penalty than in the P6 and Pentium. This is because the instruction queue is only eight instructions long, and a flush of the queue is likely to incur only a 1- or 2-cycle penalty.

Michael, a software engineer for MetaWare, can be contacted at miker@metaware.com.

The BPU has three special-purpose registers that are not part of the usual general-purpose registers:

- Link register (LR).
- Count register (CTR).
- Condition register (CR).

The BPU calculates and saves the return pointer for subroutine calls in the LR. The CTR contains the target address for some conditional branch instructions. The LR and CTR can be easily copied to or from any general-purpose integer register. Because the BPU has these special-purpose registers, all branching except for synchronization can be carried out independent of the integer and floating-point units. Unlike the Pentium, which has many special conditions that must be fulfilled before instructions can execute in parallel (often producing stalls), the PowerPC's BPU unit helps prevent stalls from occurring. Since the PowerPC architecture reserves these registers (LR, CTR, CR) for the branch processing unit, compilers have more integer registers available for allocation of important variables.

From the compiler's point of view, the number of fast, general-purpose registers available on a processor is a key factor in the execution speed of applications. In this respect, the PowerPC has the Pentium beat. The Pentium has only eight 32-bit integer registers: ESP and EBP are dedicated to special purposes, and other registers are implicitly destroyed by certain instructions, creating a headache for register allocation. The P6 processor (with 40 general registers and a hardware-register-renaming scheme) helps, but it doesn't solve the problem, due to the need for backward compatibility. Under the PowerPC application binary interface (ABI), the compiler has 15 general-integer registers that may be used just for local variables. On the Pentium, some optimizations such as strength reduction for array accesses simply cause too much competition for registers, and rarely pay off. The PowerPC allows you to take advantage of all the possible optimizations at your disposal.

The instruction unit actually contains the instruction queue and the BPU, providing a central control over the instructions issued to the execution units, the integer unit, and the floating-point unit. The instruction unit determines the next instruction to be fetched from the 32-byte cache and controls pipeline interlocks. It allows out-of-order execution when instructions do not depend on the result of an instruction currently executing.

The PowerPC's integer unit (IU) performs all loads, stores, and integer arithmetic. It contains an ALU and an integer exception register (XER). Most integer instructions execute in one clock cycle. Loads and stores are issued and translated in sequential order, but actual memory access can occur out of order. Synchronizing instructions are available for times when order is critical. The IU contains 32 integer registers, each 32 bits wide. 64-bit versions of the PowerPC are coming soon.

The floating point unit (FPU) is fully IEEE 754 compliant and contains 32 floating-point registers, each of which can hold either a single- or double-precision operand. The FPU can look ahead and find instructions in the queue that do not depend on unexecuted instructions and process the latter early. The FPU is pipelined, so that most instructions can be issued back to back, without stalling. Unlike the stack-style access to floating operands of the Intel chips, the floating-point registers allow true random access, so complex scheduling and swapping algorithms are not necessary to achieve good performance.

Measuring Performance

The difference in philosophies between the PowerPC and the Pentium becomes more evident as you begin to analyze code and performance. The Pentium chip places the burden of good performance squarely on the compiler writer's shoulders. The compiler writer has to be aware of all the special conditions that allow parallel execution of integer instructions in the U and V pipes on the Pentium, and the many conditions where instruction pairing isn't possible. The PowerPC lacks all these constraints and provides a lot of hardware assistance to make the job easier. Compounding the performance problem for application vendors is the fact that the same sequence of instructions won't run universally well across the family of Intel processors. For example, to gain performance on the Pentium, you should replace integer multiply instructions with a sequence of less complex shifts and adds where possible. However, on other Intel processors such as the P6, you should do just the opposite—use the integer multiply instruction *rather* than adds and shifts. On the 80486, you need to be concerned with whether an instruction is aligned so that it crosses a 32-byte boundary, while on the Pentium this makes little difference.

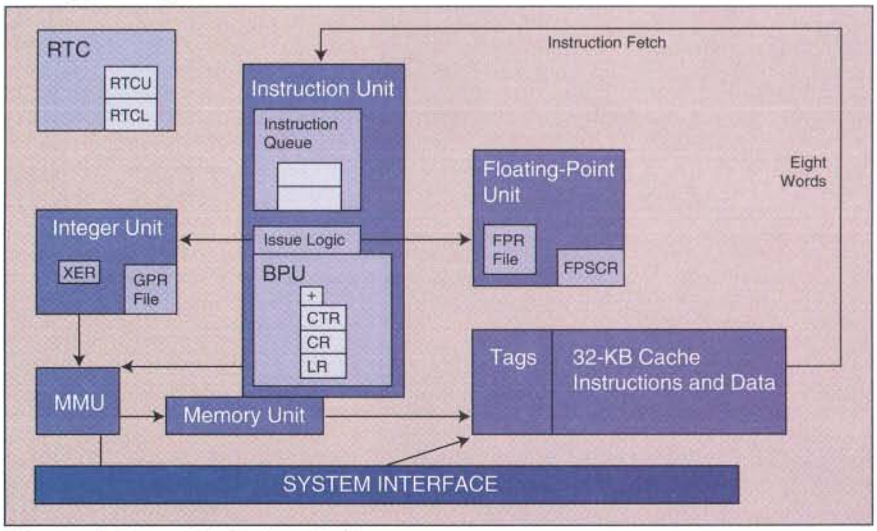


Figure 1: *PowerPC 601 block diagram.*

It's much easier to get an application that has uniform performance across the PowerPC family. A spreadsheet vendor might have to compile for the lowest common denominator on the Intel family (probably the 80486), so in many cases, customers would not get to use the power of their Pentium processors. The main thing Pentium has going for it is a huge installed base, and a lot of shrink-wrapped software for Windows/NT, DOS, and OS/2.

Although no benchmark is a real indication of how well or poorly your application will run on a given platform, the following two programs are more than just benchmarks, because people really use them in their work.

The first, Espresso, is an almost exclusively integer benchmark. Several "hot spots" dominate its execution time, one of which is the routine *massive_count* in cofactor.c. This routine is mainly a large sequence of If-Then-Else constructs that should be a natural for branch-prediction and cross-jumping optimizations. The C source code for Espresso is shown in Listing One; listings begin on page 17. Notice that in the first two loops, the bulk of the operations are of the form: *if (val & <constant>) cnt[<constant subscript>]++;*. This form shows that each operation is independent of successive ones. A good compiler will keep *val* in a register, along with the base address of the array *cnt*, and the PowerPC will fill the integer pipeline so that instructions keep executing at a one-per-cycle rate without stalls. Because the value in each condition is a constant, the BPU should be able to easily predict the need to branch or fall through. The compiler should hoist the load of the base address of *cnt* out of each If-Then-Else construct.

In assembly-language format, most PowerPC instructions use the rightmost two registers as operands and the leftmost as the destination. The code in Listing Two was generated by the MetaWare PowerPC C/C++ compiler for Solaris on PowerPC. The only surprise in Listing Two is that the test for the next If is scheduled ahead of the increment for the last one. The reason is simple: The 601 BPU is highly independent of the integer-execution unit, and the add and store instructions do not affect the condition codes in the BPU. Moving the test up avoids a stall due to a dependency on *%r10* from one instruction to the next. The distance between branch instructions is small, allowing the BPU to look ahead in the queue and easily direct the instruction stream fetch to the next set of instructions for execution. Note that the only memory references are those that are absolutely necessary. On our 60-MHz PowerPC 601 with 32 MB of RAM, Espresso executes in 48 seconds (cumulative time for all of the SPEC data input sets from the input.ref directory). It has been estimated that one in five instructions is a branch, so the importance of the BPU in the architecture really shines here. Naturally, newer versions of the PowerPC, such as the 604, would be even faster.

For comparison, Listing Three is output from the version 2.6d of the MetaWare C/C++ compiler for Unix V R4 386, on UnixWare 1.1. We compiled on a 60-MHz Pentium with the -586 switch for Pentium optimization. The ability to increment the array element to memory without loading it reduces the number of instructions needed to perform the loop. However, the real reason for this is the paucity of registers into which to load the array element. Pentium and P6 optimization lore would suggest that better instruction overlap might occur if this were more like the RISC load/store model. With the Pentium, unfortunately, this would generate an unacceptable number of spills. The P6, with its dual-integer instruction units and register renaming, may make this more feasible. Surprisingly, the Pentium is only a hair slower executing this code, managing to do all of the SPEC data sets in 50 seconds, a difference of 4 percent. The difference here appears to be in the time needed to make memory references, and the fact that the tests and branches are not independent of the integer unit. Note, however, that while this code would run the same or faster on all PowerPC variants, the same is not true of the 80x86 family, not because of clock speed, but because of differences in architecture. A 100-MHz 486 could not expect to do as well on this code, since each branch to the top of the loop would incur a 2-cycle penalty. This shows the value of branch prediction. The Pentium will suffer on integer code from its inability to pair shift and rotate instructions with variable shift counts, *mul* and *div* instructions, and some floating-point instructions. In spite of Pentium's parallel U and V pipes, some instructions don't execute in anything but the U pipe. This forces the compiler to schedule instructions so that two U pipe instructions don't occur consecutively.

Floating-Point Operations

Though less common in mainstream applications, floating point is certainly important for scientific programmers. As an exercise in portability and because I'm the author of MetaWare's Fortran compiler, I decided to do the floating-point comparison for this article using Fortran. To do this, I had to port the Fortran front end and library to Solaris PowerPC. To my surprise, the entire process took under four hours, most of which was actual compiling and linking, and the result was a compiler that passed all but three of the Fortran 77 validation suite tests on the first try. (It's been passing entirely on other platforms for years.) Solaris made the process painless.

With the compiler ported, I was able to run the familiar LINPACK benchmark. LINPACK is showing its age, but it is still used in a wide variety of real applications. One routine, SAXPY, contains a loop that is the main time consumer in the benchmark. The PowerPC version of this loop is in Listing Four. Here you can see another thoughtful aspect of the PowerPC architecture. Not many RISC architectures include instructions like floating multiply and add (or subtract). The HP PA architecture has such an instruction, but its restrictions make it unusable for LINPACK. The Pentium, for all its CISCness, includes no such instruction. The compiler has reduced the addressing computations in this loop by doing strength reduction. And of course, there's no lack of general floating-point registers to work with.

Listing Five shows how the Pentium compares. In spite of best efforts and some good optimization techniques such as loop reversal and loop unrolling, Pentium doesn't come off too well here. The major problem again is lack of registers and the rather odd floating-point stack architecture. Pentium turns in a performance of 7.616 Mflops, compared to the PowerPC's 8.58 Mflops.

Optimization Techniques

The compiler uses a number of techniques to make your code run more efficiently: common-subexpression elimination, dead-store elimination, register allocation, and global-constant propagation. The impact of these optimizations depends upon the architecture and the application code itself. For example, Listing Six shows the assembly-language output on the Pentium using the SAXPY loop without any optimization turned on.

Listing Six also shows the useless overhead in loading the address of the array element and performing the loop. The net effect is a loss in performance down to 5.317 Mflops.

The general optimizations pay off. If you compare the optimized and unoptimized code, you'll mainly see the effects of register allocation, loop unrolling, and induction elimination. But for fast code, each code generator needs to pay attention to the specific quirks of the target machine. For each architecture, the MetaWare compiler uses two phases, massage and expand, that work on the intermediate language form of the program. Here, the compiler must consider whether the machine has transcendental floating-point instructions and scaled-indexing addressing modes, and whether multiply or a series of shifts and adds is faster for that particular processor. On the PowerPC, for example, the compiler does not replace constant multiply with shifts and adds unless the final instruction sequence is no more than two instructions longer. The reason is that multiplies are fairly cheap on a PowerPC. On the Pentium, you look for things like block moves that might tie up too many registers and decrease the register pressure by combining base and index registers. On most of the architectures, these phases try to eliminate the use of a special frame-pointer register where possible and use the stack pointer instead, freeing up another general register for other purposes. With the Pentium, this is particularly important, since registers are so scarce.

A third phase for code improvement combines peephole optimization and scheduling. Each code generator has the option of both a high-level scheduling pass on the intermediate language and a low-level scheduling pass on the actual machine instructions. Because this is table driven, the code that actually does the scheduling can be the same in both cases. The tables take into account the vagaries of U and V pipe pairing on the Pentium and the pipeline stages of results on the PowerPC.

Programming for Performance

As is evident from the PowerPC design, hardware dynamic-branch prediction is beginning to supplant the older branch-delay slot design, where an instruction was moved after a branch to execute while waiting for the branch to take effect. Most chips have a finite cache or instruction queue that they can examine in order to predict the branch. If you keep the distance between a conditional branch and its target small and use expressions that are easily dynamically evaluated by something like the PowerPC's BPU, you'll increase the chance that the BPU will effectively predict whether the branch will be taken. Also, if the target and the branch fall within the cache size, your code will likely execute faster. You should separate expressions and their consumers as widely as possible, within cache limits. For example, given the expressions in Figure 2(a), consider reordering the statements to Figure 2(b). This makes it possible to complete all the operations of the first statement while processing some of the independent operations of the second, thus avoiding any possible stalls waiting for the completion of the first statement.

```
(a)  x=y*z+2.0
     z=√x
     c=π*(r*r)

(b)  x=y*z+2.0
     c=π*(r*r)
     z√x
```

Figure 2: *Reordering statements to improve performance.*

For processors like the Pentium, look through the critical regions of your code after profiling your application. If your critical loops have more than four or five heavily used variables, try to reduce that number. Since the Pentium only has a few general registers available for local variables, you can give the compiler a boost in optimizing your code if you confine the number of heavily used variables or constants to a small number in loops. For floating-point code, think about ways to break down transcendentals or floating-point divide instructions into different expressions. For example, the MetaWare compiler tries to replace *ARCSIN* with *ARCTAN2(X,SQRT((1–X)*(1+X))*, which can be done with inline code.

By breaking down complex operations into simpler ones, you'll give most processors the chance to schedule your code for faster execution. You can't always depend on your compiler to make transformations for you. Keep your floating-point expressions smaller than the size of the Pentium's floating-point stack. If you write a huge expression with more than seven or eight intermediate results, you are forcing the compiler to spill from the stack to some temporary location, or to use memory-reference instructions to complete the calculation, resulting in slower code. A little extra care in the critical regions of your program, keeping the characteristics of current processors in mind, can pay large dividends in performance.

Conclusion

With the PowerPC's price-to-performance ratio closing in on that of the Pentium, users are finding this architecture more attractive. In addition to the Macintosh, there will soon be opportunities to field applications on Solaris and OS/2 for the PowerPC. The Mac already has fairly successful DOS 80x86 emulation, enabling you to run a lot of shrink-wrapped software without sacrificing your software investment. New applications can benefit from the scalability of the PowerPC family and the new speed it offers. However, you'll need to rethink strategies in order to wring out that last ounce of performance.

DDJ

THE ABILITY TO STOP CHILD ABUSE IS NOW IN ALL OUR HANDS.

For years, child abuse has been a problem to which there were few real answers. But now there's an innovative new program designed to help stop the abuse before it can ever begin. It's already achieving unprecedented results. But we need your help where you live. Please call 1-800-C H I L D R E N today to learn more.

THE MORE YOU HELP THE LESS THEY HURT.

Ad Council

1-800-CHILDREN

Listing One

```
#include "espresso.h"
void massive_count(pcube *T){
    int *count = cdata.part_zeros;
    pcube *T1;

    /* Clear the column counts (count of # zeros in each column) */
    {
    register int i;
    for(i = cube.size - 1; i >= 0; i--)
    count[i] = 0;
    }

    /* Count the number of zeros in each column */
    {
    register int i, *cnt;
    register unsigned int val;
    register pcube p, cof = T[0], full = cube.fullset;
    for(T1 = T+2; (p = *T1++) != NULL; )
    for(i = LOOP(p); i > 0; i--)
        if (val = full[i] & ~ (p[i] | cof[i])) {
        cnt = count + ((i-1) << LOGBPI);
#if BPI == 32
        if (val & 0xFF000000) {
        if (val & 0x80000000) cnt[31]++;
        if (val & 0x40000000) cnt[30]++;
        if (val & 0x20000000) cnt[29]++;
        if (val & 0x10000000) cnt[28]++;
        if (val & 0x08000000) cnt[27]++;
        if (val & 0x04000000) cnt[26]++;
        if (val & 0x02000000) cnt[25]++;
        if (val & 0x01000000) cnt[24]++;
        }
        if (val & 0x00FF0000) {
        if (val & 0x00800000) cnt[23]++;
        if (val & 0x00400000) cnt[22]++;
        if (val & 0x00200000) cnt[21]++;
        if (val & 0x00100000) cnt[20]++;
        if (val & 0x00080000) cnt[19]++;
        if (val & 0x00040000) cnt[18]++;
        if (val & 0x00020000) cnt[17]++;
        if (val & 0x00010000) cnt[16]++;
        }
#endif
        if (val & 0xFF00) {
        if (val & 0x8000) cnt[15]++;
        if (val & 0x4000) cnt[14]++;
        if (val & 0x2000) cnt[13]++;
        if (val & 0x1000) cnt[12]++;
        if (val & 0x0800) cnt[11]++;
        if (val & 0x0400) cnt[10]++;
        if (val & 0x0200) cnt[ 9]++;
        if (val & 0x0100) cnt[ 8]++;
        }
        if (val & 0x00FF) {
        if (val & 0x0080) cnt[ 7]++;
        if (val & 0x0040) cnt[ 6]++;
        if (val & 0x0020) cnt[ 5]++;
        if (val & 0x0010) cnt[ 4]++;
        if (val & 0x0008) cnt[ 3]++;
        if (val & 0x0004) cnt[ 2]++;
        if (val & 0x0002) cnt[ 1]++;
        if (val & 0x0001) cnt[ 0]++;
        }
          }
 }

    /*
     * Perform counts for each variable:
     *    cdata.var_zeros[var] = number of zeros in the variable
     *    cdata.parts_active[var] = number of active parts for each variable
     *    cdata.vars_active = number of variables which are active
     *    cdata.vars_unate = number of variables which are active and unate
     *
     *    best -- the variable which is best for splitting based on:
     *    mostactive -- most # active parts in any variable
     *    mostzero -- most # zeros in any variable
     *    mostbalanced -- minimum over the maximum # zeros / part / variable
     */

    {
    register int var, i, lastbit, active, maxactive;
    int best = -1, mostactive = 0, mostzero = 0, mostbalanced = 32000;
    cdata.vars_unate = cdata.vars_active = 0;

    for(var = 0; var < cube.num_vars; var++) {
    if (var < cube.num_binary_vars) { /* special hack for binary vars */
        i = count[var*2];
        lastbit = count[var*2 + 1];
        active = (i > 0) + (lastbit > 0);
        cdata.var_zeros[var] = i + lastbit;
        maxactive = MAX(i, lastbit);
        }
        else{
        maxactive = active = cdata.var_zeros[var] = 0;
        lastbit = cube.last_part[var];
        for(i = cube.first_part[var]; i <= lastbit; i++) {
        cdata.var_zeros[var] += count[i];
        active += (count[i] > 0);
        if (active > maxactive) maxactive = active;
        }
    }

    /* first priority is to maximize the number of active parts */
    /* for binary case, this will usually select the output first */
    if (active > mostactive)
        best = var, mostactive = active, mostzero = cdata.var_zeros[best],
        mostbalanced = maxactive;
    else if (active == mostactive)
        /* secondary condition is to maximize the number zeros */
        /* for binary variables, this is the same as minimum # of 2's */
        if (cdata.var_zeros[var] > mostzero)
        best = var, mostzero = cdata.var_zeros[best],
        mostbalanced = maxactive;
        else if (cdata.var_zeros[var] == mostzero)
        /* third condition is to pick a balanced variable */
        /* for binary vars, this means roughly equal # 0's and 1's */
        if (maxactive < mostbalanced)
            best = var, mostbalanced = maxactive;
    cdata.parts_active[var] = active;
    cdata.is_unate[var] = (active == 1);
    cdata.vars_active += (active > 0);
    cdata.vars_unate += (active == 1);
        }
    cdata.best = best;
    }
}
```

Listing Two

```
 !137 |      if (val = full[i] & ~ (p[i] | cof[i])) {
       sli     %r8,%r11,2         !+a4  Shift i, which is %r11, left 2
       lwzx    %r10,%r8,%r12      !+a8  Add %r8(p) and %r12 to form address, load
       lwzx    %r7,%r8,%r4        !+ac  Add %r4 (cof) and %r8 (i<<2), load
       nor     %r10,%r7,%r10      !+b0  or not operation
       lwzx    %r8,%r8,%r5        !+b4  Add %r5 (full), %r8(i), load
       and.    %r8,%r8,%r10       !+b8 and of full[i] &~ (p[i] | cof[i])
                                  ! val ends up in %r8
       beq     ..LL63  !+bc
!138 |         cnt = count + ((i-1) << LOGBPI);
       sli     %r10,%r11,7        !+c0
       addi    %r6,%r10,-128      !+c4
       add     %r7,%r29,%r6       !+c8 Get base address assigned to cnt in %r7
!139 |#if BPI == 32
!140 |      if (val & 0xFF000000) {
       andis.  %r10,%r8,65280  !+cc  Test bits in val
       beq     ..LL64  !+d0           Branch  if bits not set
!141 |         if (val & 0x80000000) cnt[31]++;
       andis.  %r10,%r8,32768  !+d4  Test bits in val
       beq     ..LL65  !+d8           Branch if bits not set
       lwz     %r10,+124(%r7)  !+dc  Base address already in %r7, load element
       andis.  %r31,%r8,16384  !+e0 Scheduling places test of val ahead
                               ! to avoid integer pipeline stall
       addi    %r10,%r10,1     !+e4 add 1 to cnt[31]
       stw     %r10,+124(%r7)  !+e8 Store element to memory
!142 |         if (val & 0x40000000) cnt[30]++;
       beq     ..LL66  !+ec  Branch on pretested condition
       b       ..LL67  !+f0
       andis.  %r10,%r8,16384  !+f4 Test next bits in val
       beq     ..LL66  !+f8          Branch if not set
       lwz     %r10,+120(%r7)  !+fc  Load cnt[30]
       andis.  %r31,%r8,8192   !+100 Test next bits
       addi    %r10,%r10,1     !+104 Increment cnt[30]
             stw     %r10,+120(%r7)  !+108  Store back to memory
```

Listing Three

```
/136 :  for(i = LOOP(p); i > 0; i--)
        movl    (%edx),%eax
        andl    $1023,%eax       / 0x3ff
        testl   %eax,%eax    / initialize i, get it into %eax, test for zero
        jle     .L43
.L44:
/137 :      if (val = full[i] & ~ (p[i] | cof[i])) {
        movl    (%edx,%eax,4),%ebx   / Load cof[i] into %ebx
        movl    68(%esp),%esi        / Load index into %esi
        orl     (%esi,%eax,4),%ebx   / or together, from memory
        movl    64(%esp),%esi        / load full[i] index
        notl    %ebx                 / not of expr
        andl    (%esi,%eax,4),%ebx   / and from memory
        testl   %ebx,%ebx            / val now in %ebx
        je      .L45
/138 :          cnt = count + ((i-1) << LOGBPI);
        movl    %eax,%edi            / Copy i
        shll    $7,%edi              / Shift left
        testl   $-16777216,%ebx / 0xff000000  / Test bits in val
        lea     -128(%edi,%ecx),%esi   / Get base address into %esi
/139 : #if BPI == 32
/140 :      if (val & 0xFF000000) {
        je      .L46                       / Branch on bits not set
/141 :          if (val & 0x80000000) cnt[31]++;
        testl   $-2147483648,%ebx    / Test bits in val
        je      .L47                 / branch on bits not set
        incl    -4(%edi,%ecx)        / Increment cnt[31]
.L47:
/142 :          if (val & 0x40000000) cnt[30]++;
        testl   $1073741824,%ebx         / 0x40000000
        je      .L48
        incl    120(%esi)
.L48:
/143 :          if (val & 0x20000000) cnt[29]++;
        testl   $536870912,%ebx / 0x20000000
        je      .L49
        incl    116(%esi)
```

(continued on page 18)

(continued from page 17)

Listing Four

```
!459 ¦      do 30 i = 1,n
       addi    %r11,%r7,4       !+68 Pointer to dy(i)
       addi    %r10,%r5,4       !+6c Pointer to dx(i)
       mtctr   %r9     !+70
       addi    %r10,%r10,-8     !+74  Array bias
       addi    %r11,%r11,-8     !+78
!460 ¦         dy(i) = dy(i) + da*dx(i)
       lfs     %f12,+4(%r11)    !+7c Load dy(i)
       lfsu    %f0,+4(%r10)     !+80 Load dx(i)
       fmadds  %f0,%f0,%f13,%f12        !+84 Floating multiply and add
       stfsu   %f0,+4(%r11)     !+88 Store result into dy(i)
!461 ¦  30 continue
       addi    %r12,%r12,1      !+8c Increment loop counter
       bdnz    ..LL97  !+90   test and branch non-zero
       b       ..LL98  !+94
```

Listing Five

```
/459 :        do 30 i = 1,n
    cmpl    $2,%ecx  / Check for loop execution
    movl    68(%esp),%esi / Get pointer to dx
    movl    60(%esp),%edx /
    jle    .L82
    jmp    .L83
.L81:
.L91:
    movl    %eax,%edi

.L83:
/460 :          dy(i) = dy(i) + da*dx(i)
    flds    -4(%edx,%edi,4) / load dx(i)
    fmul    %st(1),%st     / da * dx(i)
    addl    $-2,%ecx       / Loop was reversed to count down
    lea     2(%edi),%eax
    fadds   -4(%esi,%edi,4) / Add dy(i)
    cmpl    $2,%ecx
    fstps   -4(%esi,%edi,4) / Store result to dy(i)

/461 :     30 continue
      flds    (%edx,%edi,4)  / Next loop iteration, loop unrolling
      fmul    %st(1),%st
      fadds   (%esi,%edi,4)
      fstps   (%esi,%edi,4)
      jg     .L91
      jmp    .L92
```

Listing Six

```
/459 :        do 30 i = 1,n
        movl    $1,.L98.I
        decl    %ecx
                lea     1(%ecx),%eax
               andl    %eax,%eax
               jle     .L55
               jmp     .L57
 .L57:
/460 :          dy(i) = dy(i) + da*dx(i)
        movl    .L98.I,%ecx    / I is now static, loaded from memory
        movl    24(%ebp),%edi
        flds    -4(%edi,%ecx,4)
        movl    16(%ebp),%edx  / Load index
        movl    12(%ebp),%esi  / Load address of array element
        flds    -4(%edx,%ecx,4) / Load one array element to floating stack
        fmuls   (%esi)         / Multiply
        faddp   %st,%st(1)     / Add
        fstps   -4(%edi,%ecx,4)  / Store result
/461 :    30 continue
        incl    %ecx           / Do loop book keeping
        movl    %ecx,.L98.I
        decl    %eax
        andl    %eax,%eax
        jg      .L57
```

End Listings

PowerPC 601 and Alpha 21064

Second generation RISC processors

Shlomo Weiss and James E. Smith

Just as there is more than one way to skin a cat, there is more than one way to implement RISC concepts. The PowerPC is a good example of a high-performance RISC implementation that is tuned to a specific architecture. It isn't, however, the only RISC implementation style that processor designers have used. We'll compare the PowerPC 601 to an alternative RISC architecture and implementation—the DEC Alpha 21064.

The 601 focuses on relatively powerful instructions and great flexibility in instruction processing. The 21064 depends on a very fast clock, with simpler instructions and a rigid implementation structure. Both the 601 and the 21064 have load/store architectures with 32-bit, fixed-length instructions. Each has 32 integer and 32 floating-point registers, but beyond these basic properties, they have little in common; see Table 1.

The 601 has a relatively small die size due to IBM's aggressive 0.6-micron CMOS technology with four levels of metal (a fifth metal layer is used for local interconnect); see Table 2. The cache size of each chip largely accounts for the substantial difference in the transistor count. Two striking differences appear in clock cycle and power dissipation. The 21064 is much faster, but also runs much hotter. It's well-known that CMOS's faster clock gives it more power, but even if a fast clock "wins" in performance, its higher power-consumption requirements could "lose" in usefulness—in portable PCs, for example.

Shlomo is a faculty member in the Department of Electrical Engineering/Systems at Tel Aviv University. Jim is a faculty member in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. They are the authors of POWER and PowerPC: Principles, Architecture, Implementation *(Morgan Kaufmann Publishers, 1994). Shlomo and Jim can be reached through the* DDJ *offices.*

PowerPC 601 Pipelines

All instructions for the 601 are processed in the fetch and dispatch stages. Branch and Condition Register instructions go no farther. Fixed-point and load/store instructions are also decoded in the dispatch stage of the pipe and are then passed to the FXU to be processed. Most fixed-point arithmetic and logical instructions take just two clock cycles in the FXU: one to execute and one to be written into the register file. All load/store instructions have three cycles in the FXU: address generation, cache access, and register write. This assumes a cache hit, of course.

The 601 design emphasizes getting the FXU instructions processed in as few pipeline stages as possible. This low-latency design is evident in the combining of the dispatch and decode phases of instruction processing. The effect of an instruction pipeline's length on performance is most evident after a branch, when the pipeline may be empty or partially empty.

The shorter the pipeline, the more quickly instruction execution can start again. Most of the time, the first instructions following a branch are FXU instructions (even in floating-point-intensive code), because a program sequence following a branch typically begins by loading

	PowerPC 601	Alpha 21064
Basic architecture	load/store	load/store
Instruction length	32-bit	32-bit
Byte/halfword load/store	yes	no
Condition codes	yes	no
Conditional moves	no	yes
Integer registers	32	32
Integer-register size	32/64 bit	64 bit
Floating-point registers	32	32
Floating-register size	64 bit	64 bit
Floating-point format	IEEE 32-bit, 64-bit	IEEE, VAX 32-bit, 64-bit
Virtual address	52–80 bit	43–64 bit
32/64-mode bit	yes	no
Segmentation	yes	no
Page size	4 KB	implementation specific

Table 1: *Architectural characteristics.*

	PowerPC 601	Alpha 21064
Technology	0.6-micron CMOS	0.75-micron CMOS
Levels of metal	4	3
Die size	1.09 cm square	2.33 cm square
Transistor count	2.8 million	1.68 million
Total cache (instructions + data)	32 KB	16 KB
Package	304-pin QFP	431-pin PGA
Clock frequency	50 MHz (initially)	150 to 200 MHz
Power dissipation	9 watts @ 50 MHz	30 watts @ 200 MHz

Table 2: *Implementation characteristics.*

(a)

PowerPC 601	Integer			Floating			Branch
	Load	Store	Operate	Load	Store	Operate	
Integer load						×	×
Integer store						×	×
Integer operate						×	×
Floating load						×	×
Floating store						×	×
Floating operate	×	×	×	×	×		×
Branch	×	×	×	×	×	×	

(b)

Alpha 21064	Integer			Floating			Branch	
	Load	Store	Operate	Load	Store	Operate	Integer	Floating
Integer load			×			×		
Integer store			×					
Integer operate	×	×		×		×	×	
Floating load			×			×		
Floating store						×		
Floating operate	×		×	×	×			×
Integer branch			×					
Floating branch						×		

Table 3: *Instruction dispatch rules; (a) In the 601, three mutually compatible instructions (marked with X) may issue simultaneously; (b) in the 21064, two compatible instructions may issue simultaneously. Integer branches depend on an integer register, and floating branches depend on a floating register.*

	Integer Registers		Floating Registers	
	Read Ports	Write Ports	Read Ports	Write Ports
PowerPC 601	3	2	3	2
Alpha 21064	4	2	3	2

Table 4: *Register file ports.*

data from memory (or by preparing addresses with fixed-point instructions). Obviously, a short FXU pipeline is desirable.

In contrast, floating-point instructions are processed more slowly. FPU decoding is not performed in the same clock cycle as dispatching. The first floating-point instruction following a branch is likely to depend on a preceding load, so the extra delay in the floating-point pipeline will not affect overall performance significantly. This extra delay reduces the interlock between a floating load and a subsequent dependent floating-point instruction to just one clock cycle.

The buffer at the beginning of the FPU can hold up to two instructions; the second buffer slot is the decode latch, where instructions are decoded. In the FXU pipeline, there is a one-instruction decode buffer that can be bypassed. The decode buffers provide a place for instructions to be held if one of the pipelines blocks due to some interlock condition or an instruction that consumes the execute stages for multiple cycles. By getting instructions into the decode buffers when a pipeline is blocked, the instruction buffers are allowed to continue dispatching instructions (especially branches) to nonblocked units.

21064 Pipelines

The 21064 pipeline complex is composed of three parallel pipelines: fixed-point, floating-point, and load/store. The pipelines are relatively deep, and the integer and load/store pipes are the same length. These are the stages that an instruction may go through:

1. F, Fetch. The instruction cache is accessed, and two instructions are fetched.
2. S, Swap. The two instructions are directed to either the integer or the floating-point pipeline, sometimes swapping their positions, and branch instructions are predicted.
3. D, Decode. Instructions are decoded in preparation for issue—the opcode is inspected to determine the register and resource requirements of each instruction. Unlike IBM processors, registers are not read during the decode stage.
4. I, Issue. Instructions are issued and operands are read from the registers. The register and resource dependencies determine if the instruction should begin execution or be held back. After the issue stage, instructions are no longer blocked in the pipelines, and can therefore be completed.
5. A, ALU stage 1. Integer adds, logicals, and short-length shifts are executed. Their results can be immediately bypassed back, so these appear to be single-cycle instructions. Longer-length shifts are initiat-

ed in this stage, and loads and stores do their effective-address add.

6. B, ALU stage 2. Longer-length shifts complete and their results are bypassed back to ALU 1, so these are two-cycle instructions. For loads and stores, the data cache tags are read. Loads also read cache data.
7. W, Write stage. Results are written into the register file. Cache hit/miss is determined. Data from store instructions that hit is stored in a buffer. It will then be written into the cache during a cycle with no loads.

The 21064 integer pipeline relies on a large number of bypasses to achieve high performance. In a deep pipeline, bypasses reduce apparent latencies. There are a total of 38 separate bypass paths.

Floating-point instructions pass through F, S, D, and I stages just like integer instructions. Floating-point multiply and add instructions are performed in stages F through K. The floating-point divide takes 31 cycles for single precision and 61 cycles for double precision.

Dispatch Rules

The dispatch rules in the 601 are quite simple. The architecture has three units—Integer (or Fixed Point), Floating Point, and Branch—that can process instructions simultaneously. Integer operate instructions and all loads and stores go to the same pipeline (FXU), and only one instruction of this category may issue per clock cycle.

The 21064's swap corresponds to the 601's dispatch. Instructions issue two stages later. In the 21064, instructions must issue in their original program order, and dispatch (that is, the swap stage) helps to enforce this order. A pair of instructions belonging to the same aligned doubleword (or "quadword" in DEC parlance) can issue simultaneously. Consecutive instructions in different doublewords may not dual-issue, and if two instructions in the same doubleword cannot issue simultaneously, the first in the program sequence must issue first.

The 21064 implements separate integer and load/store pipelines, and several combinations of these instructions may be dual-issued (with the exception of integer operate/floating store, and floating operate/integer store). The separate load/store unit requires an extra set of ports to both the integer and floating register files. The load/store ports are shared with the Branch Unit, which has access to all the registers because the 21064 architecture has no condition codes, and branches may depend on any integer or floating register. Consequently, branches may not be issued simultaneously with load or store instructions.

There are significant differences in the way the PowerPC and Alpha architectures handle branches

Table 3 summarizes the dispatch rules for both chips. In the 601 table, an X in the corresponding row/column indicates that two instructions may simultaneously issue. For three instructions, all three pairs must have Xs. In the 21064 table, two instructions with an X may simultaneously issue.

The ability of the 21064 to dual-issue a load with an integer-operate instruction is a definite advantage over the 601. Many applications (not to mention the operating system) use very little floating point; the 21064 can execute these apps with high efficiency, but the 601 can execute only one integer instruction per clock cycle (while its FPU sits idle).

Register Files

The 21064 and 601 have register files with almost the same number of ports; see Table 4. Both start with one write and two read ports to service operate instructions. The 21064 provides an additional pair of read/write ports for load/store unit data. Branches share the load/store register ports, which brings the count up to 3R/2W for both integer and floating-register files. One additional integer read port is needed to get the address value for stores and loads. Doing an integer store in parallel with an integer operate involves an extra integer read port, but not allowing a register-plus-register addressing mode saves a register-read port.

The 601's one write and two read ports for operate instructions are fortified by an additional integer read port for single-cycle processing of store with index instructions, which read three registers (two for the effective address, one for the result). An extra integer write port allows the result of an operate instruction and data returned from the cache to be written in the same clock cycle. The same consideration accounts for two write ports in the floating-register file. The three floating-point read ports accommodate the combined floating multiply/add instruction.

Data Caches

The 21064 uses separate instruction and data caches. The data caches are small, (8 KB) direct-mapped data caches designed for very fast access times; see Figure 1(a). The address add consumes one clock cycle. During the next clock cycle, the Translation Lookaside Buffer (TLB) is accessed and the cache data and tag are read. In a direct-mapped cache this is easy because only one tag must be read, and the data, if present, can only be in one place. The TLB address translation completes in the third cycle, and the tag is compared with the upper address bits. A cache hit or miss is determined about halfway through this clock cycle. The data are always delivered to the registers as an aligned, 8-byte doubleword. Alignment, byte selecting, and the like must be done with separate instructions.

In the 601, the unified data/instruction cache is much larger—32 KB—and is 8-way set associative, yielding a higher hit rate than the 21064. Figure 1(b), shows how much more "work" the 601 does in a clock cycle. It does an address add and the cache directory/TLB lookup in the same cycle. During the next cycle, it accesses the 32-byte-wide data memory and selects and aligns the data field.

(a)
```
double x[512], y[512];

for (k = Ø; k < 512; k++)
    x[k] = (r*x[k] + t*y[k]);
```

(b)
```
                                      # r1 points to x
                                      # r2 points to y
                                      # r6 points to the end y
                                      # fp2 contains t
                                      # fp4 contains r
                                      # r5 contains the constant 1
LOOP:   ldt     fp3 = y(r2,Ø)         # load floating double
        ldt     fp1 = x(r1,Ø)         # load floating double
        mult    fp3 = fp3,fp2         # floating multiply double  t*y
        addq    r2  = r2,8            # bump y pointer
        mult    fp1 = fp1,fp4         # floating multiply double, r*x
        subq    r4  = r2,r6           # subtract y end from current pointer
        addt    fp1 = fp3,fp1         # floating add double, r*x+t*z
        stt     x(r1,Ø) = fp1         # store floating double to x(k)
        addq    r1  = r1,8            # bump x pointer
        bne     r4,LOOP               # branch on r4 ne Ø
```

***Example 1:** Alpha 21064 pipelined processing example. (a) C code; (b) assembly code.*

```
                    1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22

ldt  fp3=y(r2.Ø)    F  S  D  I  A  B  W
ldt  fp1=X(r1.Ø)    F  .  S  D  I  A  B  W
mult fp3=fp3.fp2          F  S  D  .  I  F  G  H  K  I  W
addq r2=r2.8              F  S  D  .  I  A  B  W
mult fp1=fp1.fp4             F  S  .  D  I  F  G  H  J  K  W
subq r4=r2.r6                F  S  .  D  I  A  B  W
addt fp1=fp3.fp1                F. S  D  .  .  .  .  .  I  F  G  H  J  K  W
stt  x(r1.Ø)=fp1                F. S  D  .  .  .  .  .  .  .  .  .  I  A  B  W
addq r1=r1.8                          F  S  .  .  .  .  .  .  .  .  .  D  I  A  B  W
bne  r4.loop                          F  S  .  .  .  .  .  .  .  .  .  D  I  A  .  .
ldt  fp3=y(r2.Ø)                            F  .  .  .  .  .  .  .  .  S  D  I  A  B
ldt  fp1=x(r1.Ø)                            F  .  .  .  .  .  .  .  .  .  S  D  I  A
```

***Example 2:** 21064 pipeline flow for loop example.*

```
                                                        Issue time

LOOP:  ldt   fp3 = y(r2,Ø)       # load y[k]                Ø
       ldt   fp1 = x(r1,Ø)       # load x[k]                1
       ldt   fp7 = y(r2,8)       # load y[k+1]              2
       ldt   fp5 = x(r1,8)       # load x[k+1]              3
       mult  fp3 = fp3,fp2       # t*y[k]                   4
       ldt   fp11 = y(r2,16)     # load y[k+2]              4
       mult  fp1 = fp1,fp4       # r*x[k]                   5
       ldt   fp9 = x(r1,16)      # load x[k+2]              5
       mult  fp7 = fp7,fp2       # t*y[k+1]                 6
       ldt   fp15 = y(r2,24)     # load y[k+3]              6
       mult  fp5 = fp5,fp4       # r*x[k+1]                 7
       ldt   fp13 = x(r1,24)     # load x[k+3]              7
       mult  fp11 = fp11,fp2     # t*y[k+2]                 8
       addq  r2  = r2,32         # bump y pointer           8
       mult  fp9 = fp9,fp4       # r*x[k+2]                 9
       subq  r4  = r2,r6         # remaining y size         9
       mult  fp15 = fp15,fp2     # t*y[k+3]                1Ø
       mult  fp13 = fp13,fp4     # r*x[k+3]                11
       addt  fp1 = fp3,fp1       # r*x[k]+t*y[k]           12
       addt  fp5 = fp7,fp5       # r*x[k+1]+t*y[k+1]       13
       addt  fp9 = fp11,fp9      # r*x[k+2]+t*y[k+2]       15
       stt   x(r1,Ø) = fp1       # store x[k]              16
       addt  fp13= fp15,fp13     # r*x[k+3]+t*y[k+3]       17
       stt   x(r1,8) = fp5       # store x[k+1]            17
       stt   x(r1,16) = fp9      # store x[k+2]            19
       stt   x(r1,24) = fp13     # store x[k+3]            21
       addq  r1  = r1,32         # bump x pointer          22
       bne   r4,LOOP             # next loop               22
LOOP:  ldt   fp3 = y(r2,Ø)       # next iteration          23
```

***Example 3:** Example loop, unrolled for the Alpha 21064.*

The 601 gets more done in fewer stages, but the 21064's clock cycle is about a third to a fourth the length of the 601's. Consequently, the 601's two clock cycles take much longer than the 21064's three cycles.

Example of Pipeline Flow

Example 1 shows a For loop in C and its corresponding 21064 assembly-language code. Note in this and subsequent examples that the notation, bit numbering and assembly language do not conform to that of Alpha; they have been modified to be consistent with PowerPC notation. Example 2 is the 21064 pipeline flow for the example loop. It shows in-order issue, dual-issue for aligned instruction pairs, and the relatively long six-clock-period floating-point latency. After the I stage, instructions never block.

The importance of the swap stage is clear from the first two instructions, which cannot dual-issue because both are loads. The second instruction is held for one cycle while the first moves ahead. The first dual-issue occurs for the first *addq-mult* pair. Because *mult* is the first instruction in the doubleword, *addq* must wait, even though no dependencies hold it back. The sequence of dependent floating-point instructions paces instruction issue for most of the loop. Note that the floating store issues in anticipation of the floating-point result. It waits only four—not six—clock periods for the result so that it reaches its write stage just in time to have the floating-point result bypassed to it.

A bubble follows the predicted branch at the end of the loop. Because other instructions in the pipeline are blocked, however, by the time the *ldt* following the branch is ready to issue, the bubble is "squashed." That is, if the instruction ahead of the bubble blocks and the instruction behind proceeds, the bubble is squashed between the two and eliminated.

Overall, the loop takes 16 clock periods per iteration in steady state. (The first *ldt* passes through I at time 4; during the second iteration, it issues at time 20.) In comparison, the 601 takes six (longer) clock periods.

Floating-point latencies are a major performance problem for the 21064 when it executes this type of code. Also, in-order issue prevents the loops from "telescoping" together as they would in the 601—there is very little overlap among consecutive loop iterations, and the small amount that occurs is mostly due to branch prediction. Each parallelogram in Figure 2 illustrates the general shape of the pipeline flow for a single loop iteration.

In the 601, the branch processor eliminates the need for branch prediction, and the out-of-order dispatch, along with multiple buffers placed at key points, telescopes the loop iterations. Telescoping in the 601 is limited by the lack of store buffer in the FPU, which other implementations may choose to provide. The RS/6000, for example, has register renaming, deeper buffers, and more bypass paths; it achieves much better telescoping than the 601.

Software pipelining or loop unrolling are likely to provide much better performance for a deeply pipelined implementation like the 21064. The DEC compilers unroll loops. Example 3 shows the unrolled version of Example 2. The example loop is unrolled four times. The clock period at which instructions pass through the I stage is shown in the right-hand column. Now, in steady state, four iterations take 23 clock periods (about six per iteration), more than three times better than the rolled version. Unrolling also emphasizes the performance advantage of dual-issue.

Loop unrolling also improves the performance of the 601, as Example 4 shows. After dispatching in the 601, instructions may be held in a buffer or in the decode stage if the pipeline is blocked. Hence, we show FXU and FPU decode time, and BU execute time (which is the same cycle in which a branch is decoded).

Assume that the loop body is aligned in the cache sector. Eight instructions are fetched, and instruction fetching can keep the instruction buffer full until time 2; after that, the cache is busy with load instructions. The instruction queue becomes empty and the pipeline is starved for instructions, but these cannot be fetched until time 9, when the cache finally becomes available. At this time, the six remaining instructions of the cache sector are fetched (the first two were fetched at time 2).

The unrolled loop (four iterations) takes 20 clock cycles (five clock cycles per loop iteration versus six in the rolled version).

Branch Instructions

There are significant differences in the way the PowerPC and Alpha architectures handle branches; see Figure 5. The PowerPC has a special set of registers designed to implement branches. Conditional branches may test fields in the Condition Code Register and the contents of a special register, the Count Register. A single branch instruction may implement a loop-closing branch whose outcome depends on both the Count Register and a Condition Code value. Comparison instructions set fields of the Condition Code Register explicitly, and most arithmetic and logical instructions may optionally set a condition field by using the record bit.

In the Alpha, conditional branches test a general-purpose register relative to zero or to odd or even. Thus, a test can be performed on the result of any instruction. Comparison instructions leave their result in a general-purpose register.

Certain control-transfer instructions save the updated program counter and use it as a subroutine return address. In the Alpha, these are special jump instructions that save the return address in a general-purpose register. In the PowerPC, this is done in any branch by setting the Link (LK) bit to 1, and saving the return address in the Link Register.

The Alpha also implements a set of conditional move instructions that move a value from one register to another, but only if a condition, similar to the branch condition, is satisfied. These conditional moves eliminate branches in many simple, conditional code sequences; see Example 5. A simple If-Then-Else sequence is given in Example 5(a). A conventional code sequence appears in Example 5(b); the timing shown is for the best-case path, assuming a correct prediction. Example 5(c) uses a conditional move. While the load is being done, both shifts can essentially be performed for free. The *shift 4* is tentatively placed in register *r3* to be stored to memory. If the test of *a* is True, then the conditional move to *c* replaces the value in *r3* with the *shift 2* results. The total time is shorter than the branch implementation (even in the best case) and does not depend on branch prediction.

In general, branch target addresses are determined in the following ways:

- Adding a displacement to the program counter (PC relative). Available in both architectures.
- Absolute. Available only in the PowerPC, where the displacement is interpreted as an absolute address if the Absolute Address (AA) bit is set to 1.
- Register indirect. Available for instructions not shown in Figure 3. These are the XL-form conditional branches in the PowerPC and jump instructions in

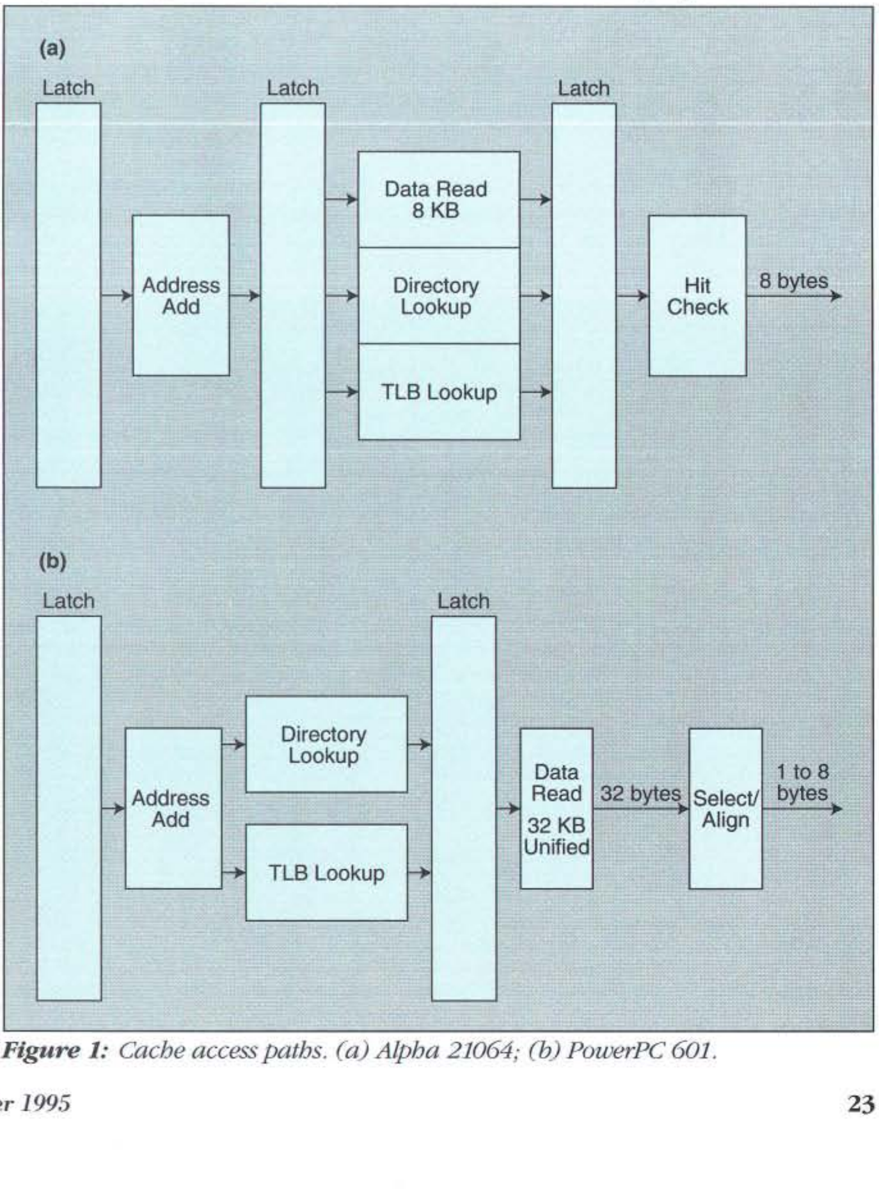


Figure 1: *Cache access paths. (a) Alpha 21064; (b) PowerPC 601.*

```
                                                            Instr. FXU    FPU    BU
                                                            fetch  decode decode exec.
                                                            time   time   time   time
      # CTR = 128 (loop count/4)
LOOP: lfs    fpØ = y(r3,2Ø52)   # load y[k]                   Ø      1
      lfs    fp4 = y(r3,2Ø56)   # load y[k+1]                 Ø      2
      lfs    fp6 = y(r3,2Ø6Ø)   # load y[k+2]                 Ø      3
      fmuls  fpØ = fpØ,fp1      # t*y[k]                      Ø             4
      lfs    fp8 = y(r3,2Ø64)   # load y[k+3]                 Ø      4
      fmuls  fp4 = fp4,fp1      # t*y[k+1]                    Ø             5
      lfs    fp2 = x(r3,4)      # load x[k]                   Ø      5
      fmuls  fp6 = fp6,fp1      # t*y[k+2]                    Ø             6
      lfs    fp5 = x(r3,8)      # load x[k+1]                 2      6
      fmuls  fp8 = fp8,fp1      # t*y[k+3]                    2             7
      lfs    fp7 = x(r3,12)     # load x[k+2]                 9     1Ø
      fmadds fpØ = fpØ,fp2,fp3  # r*x[k] + t*y[k]             9            11
      lfs    fp9 = x(r3,16)     # load x[k+3]                 9     11
      fmadds fp4 = fp4,fp5,fp3  # r*x[k+1] + t*y[k+1]         9            12
      fmadds fp6 = fp6,fp7,fp3  # r*x[k+2] + t*y[k+2]         9            13
      fmadds fp8 = fp8,fp9,fp3  # r*x[k+3] + t*y[k+3]         9            14
      stfs   x(r3+4) = fpØ      # store x[k]                 1Ø     14     15
      stfs   x(r3+8) = fp4      # store x[k+1]               1Ø     15     16
      stfs   x(r3+12) = fp6     # store x[k+2]               1Ø     16     17
      stfsu  x(r3=r3+16) = fp8  # store x[k+3]               1Ø     17     18
      bc     LOOP,CTR$\neq$Ø    # dec CTR, branch if CTR ≠ Ø 11                   15
LOOP: lfs    fpØ = y(r3,2Ø52)   # load y[k]                  2Ø     21
```

***Example 4:** Example loop, unrolled for the PowerPC 601. FXU instructions are dispatched and decoded in the same clock cycle.*

```
(a)  if (a == 1) c = b << 2;
     else c = b << 4;

(b)                                                              Issue time
                                    # initially, assume
                                    # r1 contains b,
                                    # r7 points to a,
                                    # r8 points to c.
             ldl    r2 = a(r7,Ø)    # load a from memory            Ø
             cmpeq  r5 = r2,1       # test a                        3
             beq    r5,SHFT2        # branch if a==1                4
                                    # assume taken
             sll    & r3 = r1,4     # shift b << 4
             br     & STORE         # branch uncond &
     SHFT2:  sll    & r4 = r1,2     # shift b << 2
     STORE:  stl    & r3 = c(r8,Ø)  # store c & 6

(c)                                                              Issue time
                                    # initially, assume
                                    # r1 contains b,
                                    # r7 points to a,
                                    # r8 points to c.
             ldl    & r2 = a(r7,Ø)  # load a from memory            Ø
             sll    & r3 = r1,4     # shift b << 4                  1
             sll    & r4 = r1,2     # shift b << 2                  2
             cmpeq  & r5 = r2,1     # test a                        3
             cmov   & r3 = r4,r5    # conditional move to c         4
             stl    & r3 = c(r8,Ø)  # store c                       4
```

***Example 5:** Alpha 21064 conditional-move example. (a) C code; (b) assembly code with conditional branch; (c) assembly code with conditional move.*

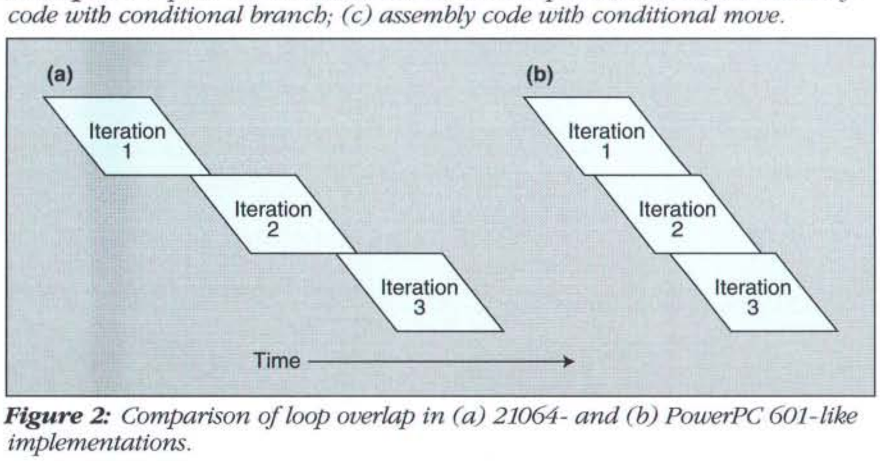


***Figure 2:** Comparison of loop overlap in (a) 21064- and (b) PowerPC 601-like implementations.*

the Alpha. General-purpose registers in the Alpha are used, and the Count Register and Link Register are used in the PowerPC.

Both processors predict branches to reduce pipeline bubbles. The 601 uses a static branch prediction made by the compiler. Also, as a hedge against a wrong prediction, the 601 saves the contents of the instruction buffer following a branch-taken prediction until instructions from the taken path are delivered from memory; thus, the instructions on the not-taken path are available immediately if a misprediction is detected.

The 21064 implements dynamic branch prediction with a 2048-entry table; one entry is associated with each instruction in the instruction cache. The prediction table updates as a program runs and contains the outcome of the most recent execution of each branch. This predictor is based on the observation that most branches are decided the same way as on their previous execution. This is especially true for loop-closing branches.

This type of prediction does not always work well for subroutine returns, however. A subroutine may be called from a number of places, so the return jump is not necessarily the same on two consecutive executions. The 21064 has special hardware to predict the target address for return-from-subroutine jumps. The compiler places the lower 16 bits of the return address in a special field of the jump-to-subroutine instruction. When this instruction is executed, the return address is pushed on a four-entry prediction stack, so return addresses can be held for subroutines nested four deep. The stack is popped prior to returning from the subroutine, and the return address is used to prefetch instructions from the cache.

Conditional-Branch Pipeline Flow

We are now ready to step through the pipeline flow for the Alpha conditional branches; see Figure 4.

The swap stage of the pipeline examines instructions in pairs. After the branch instruction is detected and predicted, it takes one clock cycle to compute the target address and begin fetching, which may lead to a one-cycle bubble in the pipeline. The pipeline is designed to allow squashing of this bubble. In the case of a simultaneous dispatch conflict, as in Figure 4(a), the instruction preceding the branch must be split from it anyway, so the branch instruction waits a cycle and fills in the bubble naturally. If the pipeline stalls ahead of the branch, the bubble can be squashed by having an instruction behind the branch move up in the pipe. If the bubble is squashed and the prediction is correct, the branch effectively becomes a zero-cycle branch.

Figure 4(b) shows the incorrect-prediction case. The branch instruction registers are read during issue stage. During the A stage, the register can be tested and the correctness of the prediction determined quickly enough to notify the instruction-fetch stage if there is a misprediction. Then, the correct path can be fetched in the next cycle. As a result, four stages of the pipeline must be flushed if the prediction is incorrect. For the jump-to-subroutine instruction, the penalty for a misprediction is five cycles.

For branches, the biggest architectural difference between the Alpha and the PowerPC is that the Alpha uses general-purpose registers for testing and subroutine linkage, while the PowerPC uses special-purpose registers held in the Branch Unit. This allows it to execute branch instructions in the Branch Unit immediately after they are fetched. In fact, the PowerPC looks back in the instruction buffer so that it can execute, or at least predict, branches while they are being fetched. The Alpha implementation, in contrast, must treat branch instructions like the other instructions. They are decoded in the D pipeline stage, read registers in I, and executed in the A stage.

Table 5 compares the branch penalties for integer-conditional branches (far more common than floating-point branches). The penalties are expressed as a function of the number of instructions (distance) separating the condition determining instruction (compare) and the branch from the correctness of the prediction. The compare-to-branch instruction count is significant only in the 601, however. Instruction cache hits are assumed.

In the 21064, correctly predicted branches usually take no clock cycles. They take one clock cycle when a bubble created in the swap stage is not later squashed. The 601 has a zero-cycle branch whenever there is enough time to finish the instruction that sets the condition code field prior to the branch and to fetch new instructions. This may take two clock cycles: one to execute the compare instruction, and one to fetch instructions from the branch target. This second clock cycle may be saved when a branch is mispredicted but is resolved before overwriting the instruction buffer; instructions may be dispatched from the buffer right after determining that the branch was not taken. With a two-instruction distance, the 601 has a zero-cycle branch even if it was mispredicted; the 21064 always depends on a prediction, regardless of the distance.

The PowerPC requires fewer branch predictions in the first place; see Table 6. In the 601, all loop-closing branches that use the CTR register do not have to be predicted; in the Alpha these are ordinary conditional branches, although loop-closing branches are easily predictable. A subroutine return must read an integer register in the Alpha, so these branches are predicted via the return stack. The PowerPC can execute return jumps immediately in the Branch Unit; there is no need for prediction.

Tables 5 and 6 show that accurate branch prediction is much more critical in the 21064. Not only does the 21064 predict more of the branches, the penalties tend to be higher when it is wrong. For this reason, the 21064 has much more hardware dedicated to the task—history bits and the subroutine return stack. The Alpha architecture also reduces the penalty for a misprediction by having branches that always test a register against zero; testing one register against another would likely take an additional clock cycle.

Some doubt the PowerPC method of using special-purpose registers for branches because they present a potential bot-

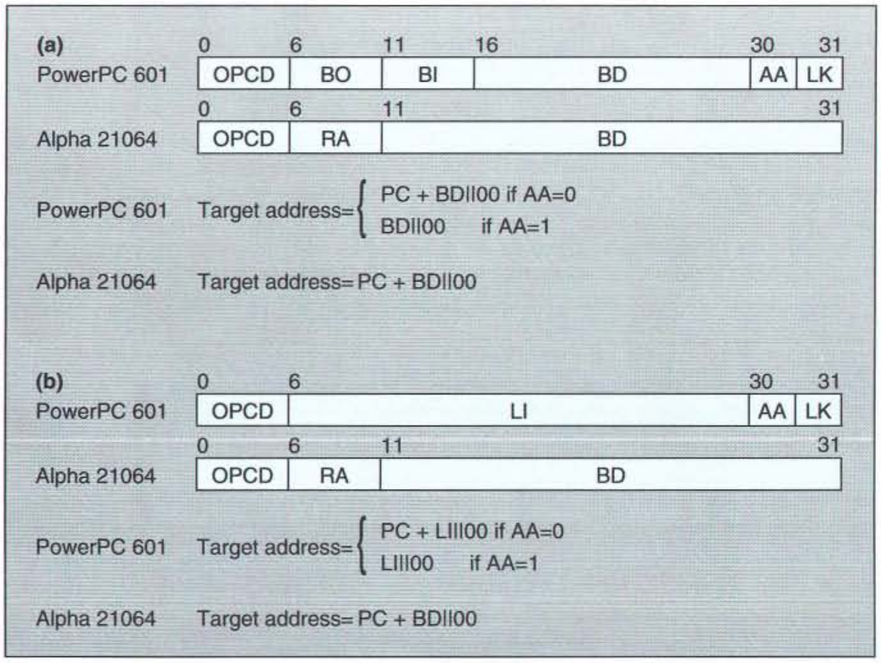

(a)
PowerPC 601
0 6 11 16 30 31
OPCD BO BI BD AA LK
Alpha 21064
0 6 11 31
OPCD RA BD
PowerPC 601 Target address= PC + BD||00 if AA=0; BD||00 if AA=1
Alpha 21064 Target address= PC + BD||00
(b)
PowerPC 601
0 6 30 31
OPCD LI AA LK
Alpha 21064
0 6 11 31
OPCD RA BD
PowerPC 601 Target address= PC + LI||00 if AA=0; LI||00 if AA=1
Alpha 21064 Target address= PC + BD||00

Figure 3: *Branch instructions. (a) Conditional branches; (b) unconditional branches.*

(a)	0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	F	S	D	I	A	B	W							
Branch	F	.	S	D	I	A								
B			F	S	D	I	A	B	W					
C			F	.	S	D	I	A	B	W				

(b)	0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	F	S	D	I	A	B	W							
Branch	F	.	S	D	I	A								
B			F	S	D	I	X	X	X					
C			F	.	S	D	X	X	X	X				
.														
.														
.														
X							F	S	D	I	A	B	W	
Y							F	.	S	D	I	A	B	W

Figure 4: *Timing for conditional branches in the Alpha 21064. (a) Instruction flow for correct branch prediction; (b) instruction flow for incorrect branch prediction. (X means instruction is flushed as a result of branch misprediction.)*

tleneck. We think not. These registers allow many branches to be executed quickly without prediction and are important for supporting loop telescoping.

Memory Architecture and Instructions

The Alpha is a 64-bit-only architecture. The PowerPC has a mode bit, and implementations may come in either 32- or 64-bit versions; the 601 is a 32-bit version. All 64-bit versions must also have a 32-bit mode. The mode determines whether the condition codes are set by 32- or 64-bit operations.

The Alpha defines a flat, or linear, virtual-address space and a virtual address whose length is implementation dependent within a specified range. The PowerPC supports a system-wide, segmented virtual-address space in either 32- or 64-bit mode. Differences between the two modes affect the number of segments and their size, which also results in a difference in the virtual-address space (52 bits versus 80 bits).

Currently, software developers and architects seem to favor flat, virtual-address spaces, although the very large segments available in the PowerPC shouldn't present many problems. The Alpha was defined as a 64-bit architecture from the start, so developers can easily provide a flat virtual-address space. The POWER architecture, however, was defined with 32-bit integer registers that were also used for addressing. This presented the POWER architects with a dilemma: Either use a flat, 32-bit virtual-address space (which would likely be too small in the very near future) or encode a larger address in 32 bits. Such an encoding led to the segmented architecture inherited by the PowerPC. Also, and perhaps more importantly, the single, shared-address space facilitates capability-based memory-protection methods similar to those used in IBM's AS/400 computer systems.

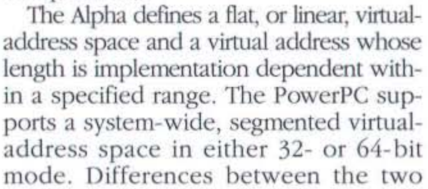


Figure 5: *Format for PAL instructions used to define operating-system primitives.*

The Alpha architecture specification does not define a page-table format. Because TLB misses are handled by trapping to system software, Alpha systems using different operating systems may have different page-table formats. Two likely alternatives are VAX/VMS and OSF/1 UNIX. A Privileged Architecture Library (PAL) provides an operating-system-specific set of subroutines for memory management, context switching, and interrupts. The Alpha instruction set includes the format in Figure 5 for PAL instructions used to define operating-system primitives.

The Call PAL instructions are like subroutine calls to special blocks of instructions, whose locations are determined by one of five different PAL opcodes. A PAL routine has access to privileged instructions but employs user-mode address translation. While in the PAL routine, interrupts are disabled to assure the atomicity of privileged operations that take multiple instructions. For example, if one instruction turns address mapping off, an interrupt should not occur until another instruction can turn it back on. The details of virtual-address translation and page-table format are a system-software issue to be defined in the context of the particular operating system using PAL functions.

Figure 6 compares the format of memory instructions. The format of instructions using the displacement-addressing mode is identical in the PowerPC and Alpha. The effective address is calculated in the same way in both architectures except for the register with the value 0, which is register 0 in the PowerPC and register 31 in the Alpha. There is no indexed addressing in the Alpha. As previously mentioned, this saves a register read port.

Another Alpha characteristic is that load and store instructions transfer only 32- or 64-bit data between a register and memory; there are no instructions to load or store 8-bit or 16-bit quantities. The Alpha architecture does include a set of instructions to extract and manipulate bytes from registers. This approach simplifies the cache interface so that it does not have to include byte-level shift-and-mask logic in the cache access path.

In Example 7, the core of a *strcpy* routine moves a sequence of bytes from one area of memory to another; a byte of zeros terminates the string. The *ldq* is a load-unaligned instruction that ignores the low-order three bits of the address; in the example, it loads a word into *r1*, addressed by *r4*. The extract byte (*extbl*) instruction uses the same address, *r4*, but only uses the three low-order bits to select one of the eight bytes in *r1*. The byte is copied into *r2*. To move the byte to *s*, the sequence begins with another load unaligned instruction to get the word containing the destination byte. The mask byte (*maskbl*) instruction uses the three low-order bits of *r3* (the address of *s*) to zero out a byte in the just-loaded *r5*. Meanwhile, the insert byte (*insbl*) instruction moves the byte from *t* into the correct byte position, also using the three low-order bits of the address in *r3*. The *bis* performs a logical OR operation that merges the byte into the correct position, and the store unaligned (*stq_u*) instruction stores the word back into *s*. The *t* and *s* pointers are incremented, the byte is checked for zero, and the sequence starts again if the byte is nonzero.

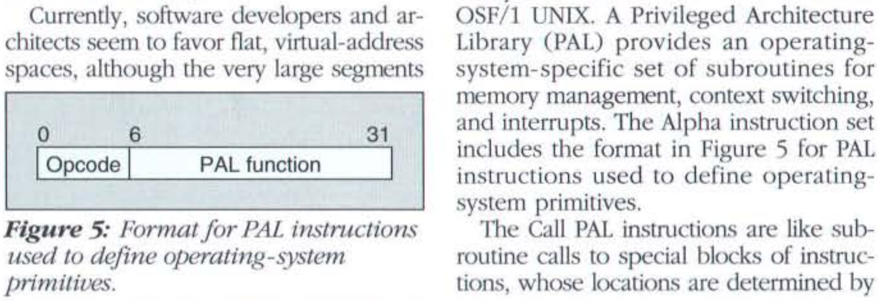

(a)
PowerPC 601
0 6 11 16 31
OPCD RT RA D
Alpha 21064
0 6 11 16 31
OPCD RT RA D
PowerPC 601
Effective address= (RA) + D if RA≠0; D if RA=0
Alpha 21064
Effective address= (RA) + D if RA≠31; D if RA=31
(b)
PowerPC 601
0 6 11 16 21 31
OPCD RT RA RB EO Rc
PowerPC 601
Effective address= (RA) + (RB) if RA≠0; D if RA=1
Alpha 21064
Register + register addressing is not available.

Figure 6: *Memory instruction format. (a) Load- and store-instruction format using register + displacement addressing. The displacement* D *is sign extended prior to addition. In Alpha,* D *is multiplied by 2^{16} if* OPCD = LDAH. RT *is the destination register. (b) load- and store- instruction format using register + register (indexed) addressing. RT is the destination register.*

	Alpha 21064		PowerPC 601	
Distance	Correct	Incorrect	Correct	Incorrect
0	0–1	4	0	2/1
1	0–1	4	0	1/0
≥2	0–1	4	0	0

Table 5: *Branch penalties.*

	Conditional Branches (non-loop-closing)	Loop-closing Branches	Subroutine Returns
PowerPC 601	Static prediction	Always zero-cycle	Always zero-cycle
Alpha 21064	Dynamic prediction	Dynamic prediction	Stack prediction

Table 6: *Prediction methods versus branch type.*

```
                                            # A string is copied from t to s
                                            # r4 points to t
                                            # r3 points to s
LOOP:   ldq_u      r1 = t(r4,0)             # load t, unaligned
        extbl      r2 = r1,r4               # extract byte from r1 to r2
        ldq_u      r5 = s(r3,0)             # load s, unaligned
        maskbl     r5 = r5,r3               # zero corresponding byte in r5
        insbl      r6 = r2,r3               # insert byte into r6
        bis        r5 = r5,r6               # logical OR places byte in r5
        stq_u      s(r3,0) = r5             # store unaligned
        addq       r4 = r4,1                # bump the t pointer
        addq       r3 = r3,1                # bump the s pointer
        bne        r6,LOOP                  # branch if nonzero byte
```

Example 7: *Alpha 21064* strcpy *function (null-terminated strings).*

Operate Instructions

The basic operations performed by both architectures are rather similar. One difference is the combined floating-point multiply-add in the PowerPC. This instruction requires three floating-point register read ports. The 21064 has three such ports but uses them for stores so that a floating-point operate can be done simultaneously with a floating point store; this can't be done in the 601.

The Alpha architecture does not have an integer-divide instruction; it must be implemented in software. Leaving out integer divides, or doing them in clever ways to reduce hardware, seems to be fashionable in RISC architectures, however, iterative dividers are cheap, and one can expect that all the RISC architectures will eventually succumb to divide instructions (some already have).

The Alpha architecture has scaled integer adds and subtracts that multiply one of the operands by 4 or 8— one of the few Alpha features that seems non-RISCy. These instructions are useful for address arithmetic in which indices of word or doubleword arrays are held as element offsets, then automatically converted to byte-address values for address calculation using the scaled add/subtracts. The PowerPC has a richer set of indexing operations embedded in loads and stores as well as the update version of memory instructions.

Conclusion

We have just seen that the PowerPC 601 and Alpha 21064 represent two distinct design philosophies. The 601 implements an instruction set containing more powerful instructions. And, it uses an implementation that provides considerable flexibility to enhance detection and exploitation of parallelism by the hardware.

Of course, this results in more-complex hardware control. The Alpha 21064 uses a very streamlined instruction set and implementation. While not appearing as clever as the 601, the simplicity of the implementation contributes to a very fast clock rate— much faster than any other commercial microprocessor.

As a final note, follow-on processors from DEC and the PowerPC consortium, the Alpha 21164 and PowerPC 604, continue the differing design philosophies. The 21164 can issue more instructions per cycle than the 21064, but its pipelines are still relatively simple, and it has a very fast clock.

The 604, on the other hand, is even more aggressive than the 601 when it comes to providing hardware mechanisms for increasing parallelism— although, as one would expect, this comes at the expense of hardware control complexity.

DDJ

ATLANTA

Maximize Your Client/Server Experience.

Whether you're seeking a contract or permanent placement, MATRIX can maximize your potential in the Atlanta marketplace. Our Staffing Specialists have the proven methods, marketplace knowledge and industry expertise to efficiently match you with your greatest opportunity. Currently, we have Client/Server positions featuring the following skills:

• Visual BASIC	• Windows NT
• Visual C++	• PeopleSoft
• OS/2	• ORACLE
• Sybase	• PowerBuilder
• Borland C++	• Clipper
• MS Windows	• Access 2.0
• UNIX/C	• Lotus Notes

Contact Lambert Chandler at:

Tel: 404-393-9933 Fax: 404-668-0384
Internet: 74740.2045@compuserve.com

MATRIX®
THE HUMAN FACTOR

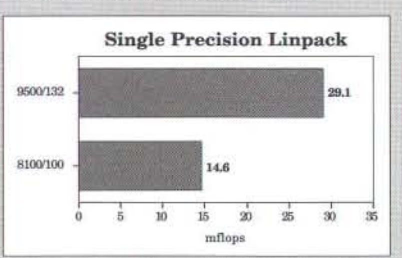
Never a fee to candidates.

LS FORTRAN

The Macintosh Leader

Engineers, scientists, and students around the world praise LS FORTRAN for its ease of use, robust code generation, and compatibility with other systems.

- ANSI FORTRAN 77
- VAX & Cray extensions
- Extensive code optimizations
- Full-featured output window
- User-friendly Mac interface
- Native PowerMac & 68K



1-800-252-6479

Language Systems Corp.
100 Carpenter Dr. • Sterling, VA 20164
(703) 478-0181 • (703) 689-9593 FAX

CIRCLE NO. 31 ON READER SERVICE CARD

metrowerks®

CodeWarrior™

C / C++ / Object Pascal

MACUSER EDITORS' CHOICE AWARD 1994 WINNER
SOFTWARE PRODUCT OF THE YEAR

Speed! Speed! Speed!
CodeWarrior compilers build your 68K and Power Mac applications with incredible speed at over 200,000 lines/min (PowerMac 8100). CodeWarrior also boasts the smallest memory and hard disk requirements for C, C++, and Object Pascal products on the Power Macintosh.

Complete Solution!
CodeWarrior allows you to develop applications for 68K, PowerPC™, Magic Cap™, and WinNT™ x86 on your favorite Mac-hosted platform. CodeWarrior's Integrated Development Environment (IDE) offers the same easy-to-use, native environment for programming in C, C++, and Object Pascal.

World-class Support and two Free Updates!
With every purchase of CodeWarrior you not only receive technical support via fax, phone, and email, but you also receive 2 free updates. Scheduled releases: May/September/January.

CodeWarrior Store Special Pricing!
As a registered user of CodeWarrior you will be eligible for special developer pricing on the following CodeWarrior supported hardware:

SONY® Magic Link™ Bundle	CodeStation™
A personal intelligent communicator package. **Includes:** • Sony Magic Link • Telebug Box (required device for debugging) • CodeWarrior Gold	A powerful new workstation from PowerComputing for Developers. **Includes:** • 16MB or 32MB of RAM • 1GB or 2GB Hard Drive • 4x CD-ROM • CodeWarrior Gold pre-loaded

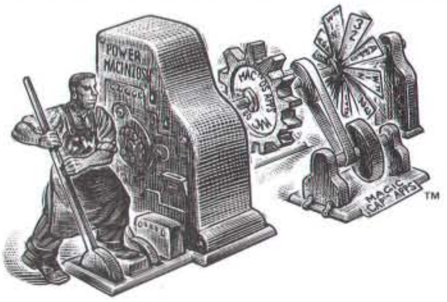
PLATFORM	LANGUAGE	GENERATING CODE FOR		GOLD $399
		PROCESSOR	OPERATING SYSTEM	
Power Mac-hosted	C/C++	PowerPC	MacOS	★
		680x0	MacOS	★
		X86	WinNT	★
	C/C++/MPW	PowerPC	MacOS	★
		680x0	MacOS	★
	Object Pascal	PowerPC	MacOs	★
		680x0	MacOS	★
	C/MPW	68349	Magic Cap	★
68K Mac-hosted	C/C++	PowerPC	MacOS	★
		680x0	MacOS	★
		X86	WinNT	★
	C/C++/MPW	PowerPC	MacOS	★
		680x0	MacOS	★
	Object Pascal	PowerPC	MacOs	★
		680x0	MacOS	★
	C/MPW	68349	Magic Cap	★
PowerPlant Application Framework				★
PowerPlant Shared Library for Power Mac				★
Source-level Debugger for Power Mac				★
Source-level Debugger for 68K Mac				★
Source-level Debugger for x86 WinNT				★
Constructor Visual Interface Editor				★
Selected Apple Developer Tools				★
On-line Documentation and much, much more				★

MacUser RATED 4½

1994 BYTE AWARD OF EXCELLENCE

AUSTRALIAN MACWORLD 1994 WORLD CLASS AWARDS

MAC LIFE GRAND PRIX 95



CIRCLE NO. 29 ON READER SERVICE CARD

To order contact Metrowerks

Metrowerks voice: (512) 305-0400 fax: (512) 305-0440 email: sales@metrowerks.com http://www.metrowerks.com

High-Performance Programming for the PowerPC

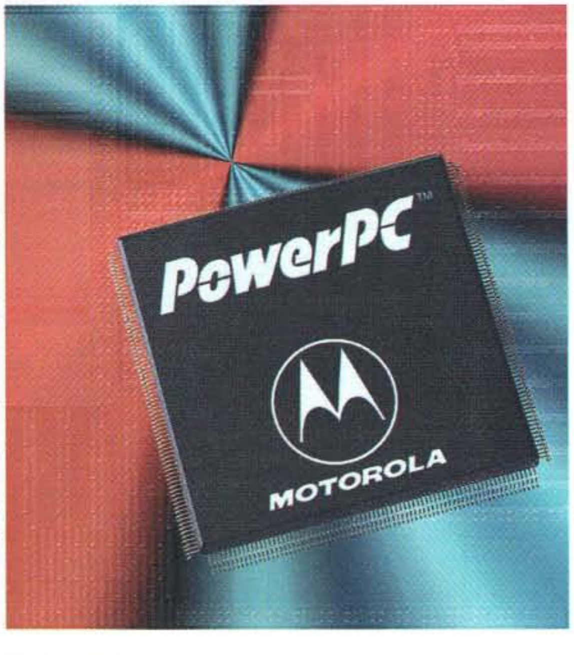
Avoid performance pitfalls when coding for Windows NT

Kip McClanahan, Mike Phillip, and Mark VandenBrink

Squeezing the best performance out of a processor requires both insight and experience. When it comes to the PowerPC microprocessor family, however, programmers are just starting to understand the architecture and each processor's implementation. The PowerPC architecture specification defines both required and optional features for any processor implementation. Each implementation of the PowerPC-architecture specification—the 601, 603, 604, and 620—may have a very different set of features. For example, cache size and type, bus width, power-management capabilities, and number of execution units can vary between each part. However, compliance with the PowerPC architecture specification ensures binary compatibility across each processor implementation.

In this article, we'll examine methods for measuring performance and techniques for improving the speed and efficiency of applications running under Windows NT for PowerPC. In doing so, we'll point out some of the pitfalls associated with Little-endian operating systems (such as Windows NT) and present an application that demonstrates the effect of byte alignment on performance. We'll also look at some optimization techniques that apply more generally to the PowerPC architecture.

Kip works in Motorola's RISC software group writing low-level PowerPC software and is the author of PowerPC Programming for Intel Programmers *(IDG Books, 1995). He can be contacted at kip_mcclanahan@risc.sps.mot.com. Mark has been hacking operating-system kernels for over ten years and is currently on the team responsible for the PowerPC port of Windows NT. He can be contacted at markv@risc.sps.mot.com. Mike is manager of the compiler and tools development group at Motorola in Austin, Texas. He can be contacted at phillip@risc.sps.mot.com.*



Measuring Performance

Perhaps the most difficult step in improving performance is simply getting started. There are several ways to analyze performance for a particular application, but it's almost always necessary to narrow the scope of the analysis to factors that can be controlled by the programmer. Performance is typically affected by:

- System hardware.
- System software.
- Application design/algorithms.
- Compiler/tools/build configuration.

System hardware issues include the size and speed of memory and disk subsystems, the type of video cards and the amount of dedicated video memory, and the type and speed of the microprocessor itself. For most developers, it is important to characterize the impact of the system hardware and software on performance, but most opportunities to improve performance lie in how the application itself it was built. System-software issues are too numerous to list, but tend to center around the operating-system and networking configuration of the computer.

Most applications are too large to examine in their entirety, but performance-analysis tools such as profilers can often identify those regions of code in which most time is spent. If profiling tools are not available, you can gain insight into any possible performance bottlenecks by thoroughly inspecting the code and answering the following questions:

- Does the application access a lot of disk-based data?
- Are the program data types integer, floating point, or both?
- Does the program frequently access large blocks of memory (such as arrays of data) or frequently allocate memory?

Improving Performance

Regardless of profiling information, a good place to start improving performance is to turn on compiler optimizations. Most modern compilers offer sophisticated code optimization that can be invoked by the user. Although such options will likely slow down compilation, the resulting application should execute much faster, often with speedups of 200 to 300 percent. Compiler optimization controls are akin to stereo-system controls, however: Just as increasing the volume to its maximum level may not be optimal for each piece of music, simply setting the default optimization flag is unlikely to yield optimal performance for every software application. Many compilers offer specific optimization flags for fine-tuning application performance. While such optimizations may not apply to enough applications to warrant inclusion in the default optimization settings, they can greatly improve the performance of a particular application.

One such option is automatic inlining of subroutines. Although not much of an optimization by itself, subroutine inlining exposes more code to the optimizer in the context in which it will be used. Of course, blindly inlining code is unwise, because replicating the subroutine body increases code size, which can actually decrease performance through loss of code locality in the memory system. However, when used in conjunction with profiling feedback, inlining small, heavily called routines often improves performance significantly for the overall application.

While selectively inlining C or C++ subroutines may improve performance, inline assembly code should be embedded with caution. When assembly code is inlined into a high-level program, the compiler must typically make conservative assumptions about register and memory usage, which can throttle many potential optimizations. But inlining critical PowerPC assembly instructions (such as synchronization primitives or status-register accesses) can improve performance. Some compilers, including those developed by Motorola, provide a set of built-in intrinsic functions that provide efficient access to low-level system instructions without incurring the performance penalty associated with inlining seemingly "random" assembly instructions.

Increased awareness of the interactions between the operating system and the microprocessor is critical to performance

Language and Design Considerations

The most significant factor affecting software performance is the design and implementation of the application itself. While algorithm design is application specific and clearly beyond the scope of this discussion, several general design considerations affect performance for virtually any PowerPC software application.

Stick to the standards. Languages such as C and C++ have well-defined standards intended to ensure the portability of source code among different development environments. Many providers of development tools offer nonstandard language extensions that differentiate their products. While often alluring to the developer, these extensions can easily tie an application to a particular tool set, and, to a lesser extent, a particular target architecture. Many of these features can significantly degrade performance on RISC microprocessors like a PowerPC.

Watch out for misalignment. Examples of such language extensions are the *__unaligned* keyword and *#pragma* pack, both of which can affect data alignment. For Little-endian implementations of an operating system such as Windows NT, misaligned accesses can lead to significant performance losses on the PowerPC architecture. Many modern microprocessors, including most PowerPC implementations, are optimized to handle properly aligned memory references, often at the expense of handling relatively infrequent misaligned references. Compilers typically will align data on its "natural" alignment boundary, where the address of the object is an exact multiple of the size of the object in bytes. However, certain programming practices, including the imprudent use of some language extensions, can create a situation where the compiler must access memory in chunks smaller than the natural size of an object. For example, if you use the *__unaligned* keyword in Microsoft C/C++ to inform a compiler that an object is misaligned, a PowerPC compiler will typically be required to load the corresponding object from memory one byte at a time. For a 32-bit integer object, this requires four memory accesses instead of one, plus three additional rotate instructions.

A similar situation can arise when the compiler is instructed to "pack" the elements of a structure via source-code *pragmas*. On older architectures, including the Intel 80x86 family, such language assertions do not necessarily affect performance. However, given the relatively high cost of servicing an alignment exception, PowerPC compilers will typically opt to generate conservative, albeit slower, code for known, potentially misaligned accesses. Removing the *__unaligned* keyword or structure-packing *pragmas* does not necessarily solve the problem. In fact, it may lead to incorrect code or even worse performance. To avoid these performance pitfalls, it's best to avoid such extensions when designing an application. The alignment example shown in Listing One (listings begin on page 38) demonstrates the performance penalties associated with the three common techniques for misalignment resolution.

Alignment exceptions can also be created through mismanagement of pointers in C and C++ programs, or by accessing data through a reference that is not of the same natural alignment as the original object; for example, accessing a series of characters or half-word objects through integer variables. For Windows NT applications, these misalignment exceptions are often hidden from the user through programmer instructions to the operating system. As the alignment example in this article demonstrates, it is clearly preferable to enable exceptions as a means of locating performance losses than to "hide" them from the user.

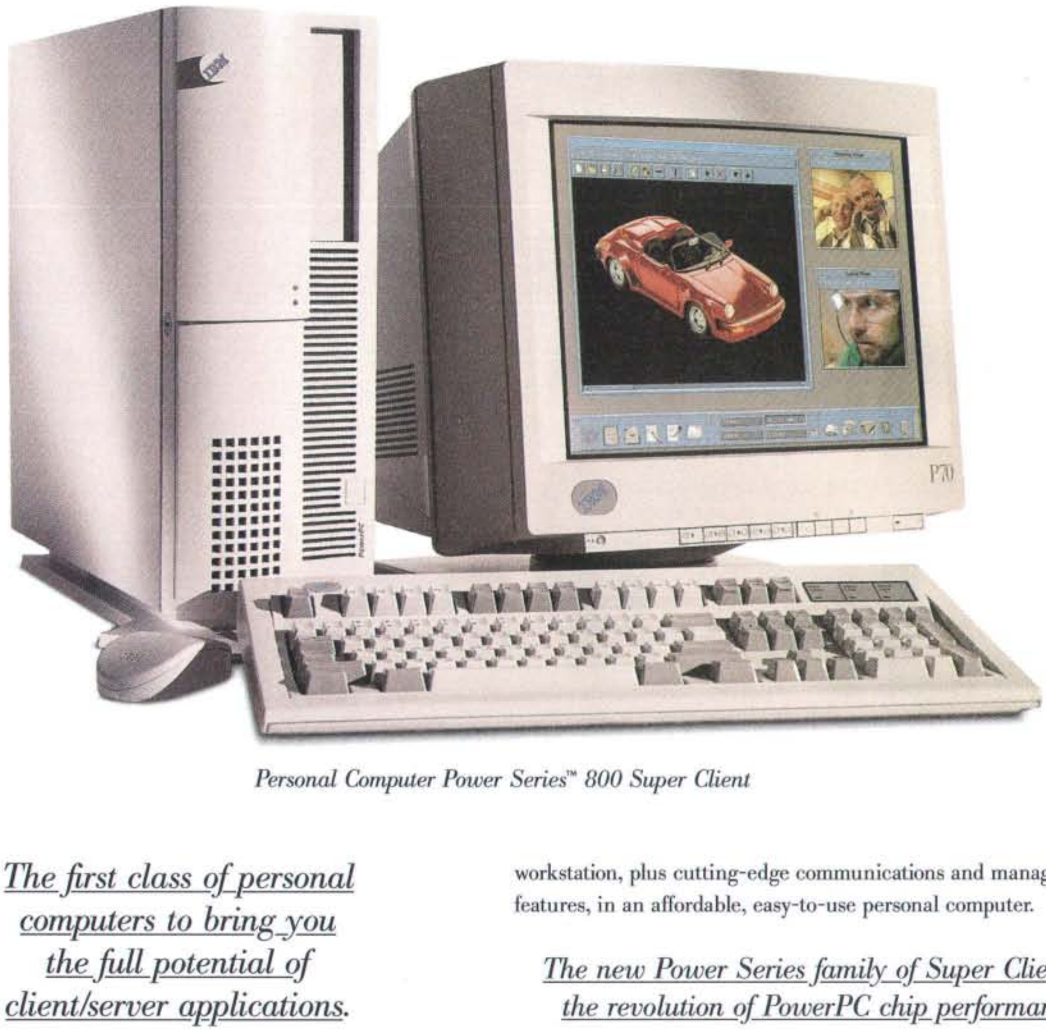
Structured Exception-Handling Issues

Structured exception handling can also adversely affect performance. While they

(continued on page 34)

For years, tapping into the full potential of client/server computing was something people could only wish for.

Introducing



Personal Computer Power Series™ 800 Super Client

The first class of personal computers to bring you the full potential of client/server applications.

For true client/server computing, there are just two benchmarks a PC needs to meet: first, deliver enough power to the desktop to handle a complex flow of information from multiple sources; second, run advanced applications using rich content formats, such as voice, video and advanced graphics.

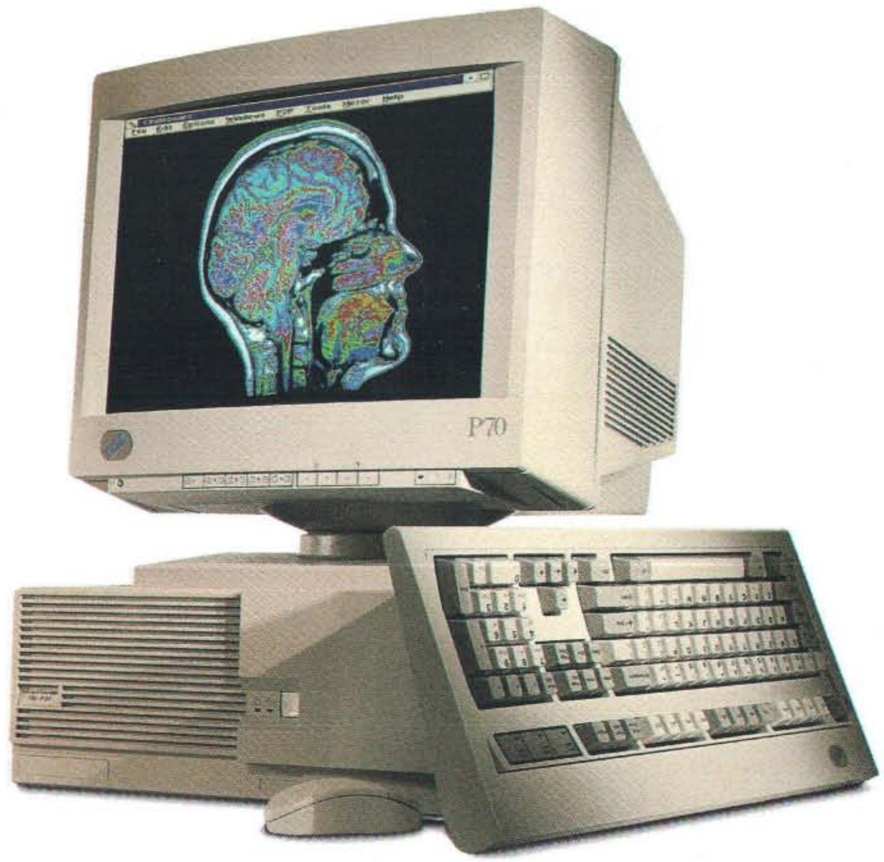
Introducing the IBM Super Clients, the new PowerPC™-based Power Series family and the PC 700, designed to maximize your current investment. Super Clients put the power of a high-end workstation, plus cutting-edge communications and management features, in an affordable, easy-to-use personal computer.

The new Power Series family of Super Clients – the revolution of PowerPC chip performance.

The open-ended performance of PowerPC RISC microprocessors, such as the fast and powerful 133MHz 604 chip, makes the Power Series family ideal for client/server environments. Combine this with your choice of robust 32-bit operating systems—OS/2® Warp Connect,[1] AIX,® Windows NT™ and Solaris®[1]—and you get the horsepower, reliability and security you need. And the Power Series family is very affordable; even with a quad-speed CD-ROM drive and 16MB memory, prices start at just $2,795 (monitor not included).[2]

[1]When available. [2]Operating systems priced separately (Power Series only). Entry-level PC Direct price. Dealer prices and product availability may vary. [3]In Canada, call 1 800 465-3299, enter #'s 45225 and 45226 for Power Series family and 45228 for PC 700. IBM, OS/2 and AIX are registered trademarks and Power Series, PowerPC, Sensory Suite, NetFinity and There is a difference are trademarks of International Business Machines Corporation. All other company and/or product names are trademarks or registered trademarks of their respective companies. © 1995 IBM Corporation.

Super Clients.



Personal Computer 700 Super Client

And Power Series systems are available with Sensory Suite™ software so you don't need add-in boards or chips to exploit graphics, music, speech or video.

PC 700 Super Client – an enhancement to your current investment.

The PC 700 delivers the blazing power of 133MHz Pentium® processors for faster data access and network communications, plus advanced multimedia digital sound capability.

And as more advanced collaborative applications become available, you will be able to enhance your system to take full advantage. PC 700 prices start at $2,200 (monitor not included).[2]

And IBM's NetFinity,™ built in to the PC 700, lets you get better control over your PC systems and lower the total cost of ownership of your client/server network.

To enroll in the IBM Power Series Developer's Toolbox Program, call us at 1 800 627-8363. Or for information on PC 700, call 1 800 IBM-4FAX[3] and enter # 8463468.

IBM.

There is a difference™

CIRCLE NO. 24 ON READER SERVICE CARD

(continued from page 30)

are an elegant and maintainable means of managing the interaction between the application and the operating system when errors or unexpected events occur, exception handlers can also restrict the compiler's ability to safely optimize code. This is true even for code not directly included in the exception handler itself. Thus, exception handlers should be carefully designed to minimize the performance impact, as follows:

- Isolate the exception-handling code as much as is practical. Since the flow of program control to an exception handler and its corresponding effect on optimization is often nonintuitive, exception handlers should not be placed in the middle of large, otherwise unrelated subroutines, particularly if they are performance critical. Since most compilers optimize a subroutine at a time, isolating the actual exception handler within a small subroutine (that is perhaps called by a larger enclosing subroutine) can help limit performance degradation.
- Avoid introducing pointers and global variables within exception-handling code. The semantics of most exception-handling language constructs typically necessitate saving and restoring "live" data across the scope of an exception handler, since the compiler often does not know which data an exception will affect. By limiting the exposure of global variables and pointers within an exception handler, the compiler can often better reduce memory accesses within the corresponding region of code.

Target-Specific Issues

Target-specific factors can often be utilized to maximize performance. Although fine-tuning for one processor or architecture at the expense of others should be avoided, a few PowerPC-specific issues should be considered to maximize PowerPC performance.

Memory Subsystem

The frequency and speed of memory accesses are almost always key factors in overall application performance. Compared to most PowerPC-processor operations, off-chip memory references are extremely expensive (particularly if the accesses are misaligned). You can often maximize the utilization of on-chip and secondary off-chip caches by carefully managing large data structures such as arrays. Since all PowerPC chip implementations utilize associative cache designs, a slight change in the "stride" of array accesses can significantly affect the effectiveness of a cache. A simple guideline is to avoid array sizes that are an exact multiple of the cache size (typically powers of 2 in the range of 8–64 KB). However, predicting cache behavior through such simple guidelines is precarious, at best, since actual dynamic reference patterns vary between applications. The guidelines' intent is to avoid allocating heavily referenced variables to the same set of cache addresses. Profiling tools can often provide feedback concerning a program's cache-utilization efficiency.

A Real-World Alignment Example

To make the following alignment example more meaningful, we'll tie it to an operating system. Because Windows NT for the PowerPC is a Little-endian operating system, it is subject to the alignment restrictions described earlier. Remember, a multibyte access performed at an address not aligned with the size of the access (known as "natural alignment") will cause an alignment exception. Table 1 shows the natural-alignment boundaries for memory accesses of various sizes.

When a Little-endian PowerPC application performs a memory access that is not aligned on its natural boundary, an alignment exception will result. When alignment issues exist on PowerPC-based systems, they can be resolved in three ways:

- If there is no support for data-alignment management, the operating system traps on the alignment exception, usually terminating the "faulting" application. This may seem the worst of all possible outcomes, but it can be very helpful during the development cycle; see the discussion that follows.
- The operating system can take the alignment exception, perform the necessary fix-ups to transparently handle the memory access, and return as if nothing had ever happened. And while abstracting the problem from both user and programmer may seem like the best solution, it is one of the worst. The transition through the alignment-exception mechanism is comparably slow and one of the worst performance killers.
- You can use the *__unaligned* type qualifier, the *#pragma pack(1)* directive, or macros on accesses with known alignment problems. Each of these techniques breaks a single multiple-byte access into individual, byte-wise accesses in order to eliminate alignment issues.

Trap and Terminate

Under Windows NT, it is possible (and the default for PowerPC Windows NT) not to have any support for misaligned data. When there is no support for misalignment and your application accesses data on an unnatural boundary, an alignment fault is generated. The Windows NT alignment fault handler is configured not to fix misaligned accesses and will display a pop-up message and terminate your application. This seems like the one situation to avoid, but in fact the operating system is doing you a favor.

The ability to trap an alignment exception and terminate the faulting application is valuable when porting code, particularly when porting Windows NT applications from 80x86 to PowerPC architectures. And the ability to detect alignment exceptions is fundamental to handling misalignment efficiently.

Operating System Fix-ups

The Windows NT kernel can be configured to perform misaligned data fix-ups upon detection of an alignment exception. This means that the alignment-exception handler must break the multibyte memory access that caused the exception into individual byte accesses, which are not constrained by alignment issues. Although this sounds like a terrific service, it is the most inefficient solution. Even if the rest of your application is well constructed, a few OS-based alignment fix-ups can bring performance to its knees. As Table 2 shows, for 5 million misaligned memory references, the

Alignment Size	Form of 32-bit Address
8-bit	xxxx xxxx
16-bit integers	xxxx xxx0
32-bit integers and single-precision FP	xxxx xx00
64-bit integers and double-precision FP	xxxx x000

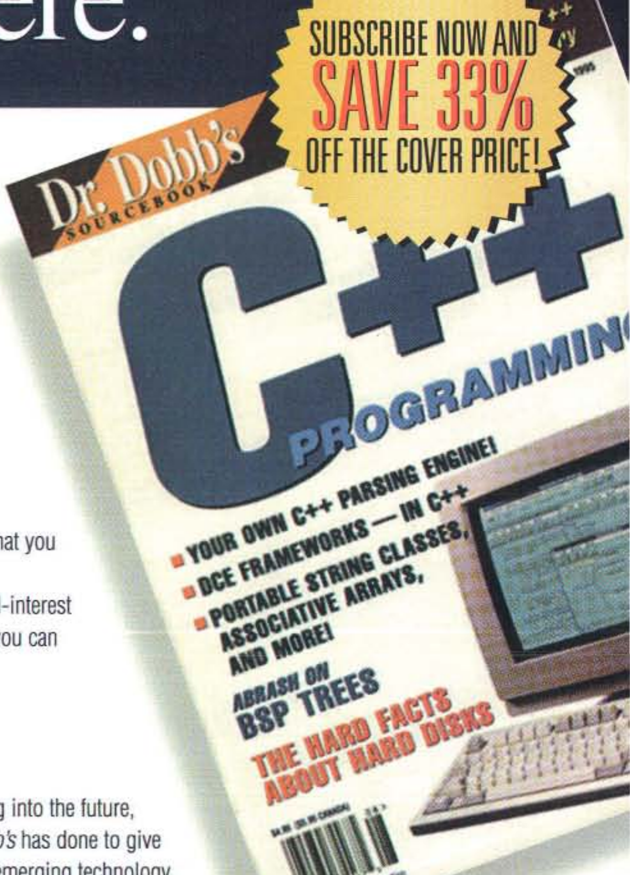
***Table 1:** Natural alignment boundaries.*

Number of Iterations	Naturally Aligned Access	OS Fix-ups	*__unaligned* Type Qualifier	*#pragma Pack(1)* for Sample BMP Structure
500,000	61 ms	2858 ms	86 ms	82 ms
1,000,000	121 ms	5720 ms	171 ms	166 ms
5,000,000	657 ms	28662 ms	852 ms	807 ms

***Table 2:** Timing values generated by Listing One. Sample tests performed on a 100-MHz 604 running Windows NT 3.51 (build 1057), compiled with Motorola's NT compiler and averaged over six runs of the program.*

Dr. Dobb's Sourcebook Series won't just show you the future, it'll take you there.

SUBSCRIBE NOW AND SAVE 33% OFF THE COVER PRICE!


Dr. Dobb's SOURCEBOOK
C++ PROGRAMMING
YOUR OWN C++ PARSING ENGINE!
DCE FRAMEWORKS — IN C++
PORTABLE STRING CLASSES, ASSOCIATIVE ARRAYS, AND MORE!
ABRASH ON BSP TREES
THE HARD FACTS ABOUT HARD DISKS

At *Dr. Dobb's Journal*, we've been putting in overtime so you don't have to. We created a magazine with the technical depth and focus to let you zoom ahead into the future of emerging technologies. While other publications seem satisfied to repeat the buzzwords and deliver broad information about new software development technologies, *Dr. Dobb's* isn't. That's why we've put our efforts toward giving you an entire magazine devoted exclusively to various **emerging technologies**–the way only *Dr. Dobb's* can do it.

Six times a year, you have access to the entire story–down to the grittiest technical detail–of a hot new technology. This is your chance to spend time with the *Dr. Dobb's* editors and writers on the topics they want to study more closely. We'll give you enough information to get your attention *and* make you an expert.

Featured Sourcebook Series Themes

Windows Programming	*Power PC Programming*
Information Highway	*Games Programming*
C/C++ Programming	*Database Programming*

Save time

When you pick up a *Sourcebook*, you know every word inside will be about the issue theme, so you won't waste time wading through pages of general-interest articles to reach the how-to technical information you want.

Save money

With the *Sourcebook Series*, you pay only for what you want: in-depth information about emerging technologies. You won't waste money on general-interest fluff that won't help you get the job done. Plus, you can take advantage of our special 33% discount.

Get there faster

Dr. Dobb's Sourcebook Series isn't about looking into the future, it's about getting there–faster. The work *Dr. Dobb's* has done to give you the *Sourcebooks* will let you approach any emerging technology we cover with a giant head start. The *Sourcebook Series* is your secret to being the first to achieve real accomplishments in developing for emerging technologies.

Don't just read about the future. Go there!

YES Enter my one-year Charter Subscription (6 issues) to Dr. Dobb's Sourcebook Series, for only $19.95. I save over 33% off the newsstand cover price.

Clip out or photocopy this form. (Mail, fax or e-mail your order.)

Name ______________________

Address ______________________

City/State/Zip ______________________

Country ______________________

❑ **Payment enclosed** ❑ **Bill me (International orders prepay in U.S. funds only.)**

Credit card no. (include all digits) ______________ Expiration date ________

Signature required ______________________

Savings based on the full one-year newsstand rate of $29.70. Foreign subscription must be prepaid in U.S. dollars drawn on a U.S. bank. Canada and Mexico: $40/year. All other foreign: $60/year (air mail). Canadian GST included (R#124771239). Please allow 4-6 weeks for delivery. sbna

TO ORDER ▼

MAIL COUPON TO:
Dr. Dobb's Sourcebook Series
P.O. Box 1525
Martinez, CA 94553-9802

▼

FAX COUPON TO:
(510) 372-8582

▼

E-MAIL:
hvessich@mfi.com
or
[cis] 71154,1113

(continued from page 34)
difference between OS handling and programmatic handling is nearly 30 seconds! Put another way, having the OS fix-up misaligned memory accesses is 43 times slower than the same number of aligned accesses.

If this solution is so slow, why is it around? Of course, you wouldn't want your released and shipping applications to terminate the first time they have an unexpected alignment fault— compatibility with 80x86 applications would be reduced significantly. OS-based fix-ups are a reasonable first-pass solution for PowerPC applications that have not specifically taken data alignment into consideration. Just as the number of 32-bit applications is slowly increasing in the 80x86 world, so will the number of PowerPC applications that make data alignment a priority.

Processes can enable OS-based alignment fix-ups using the call *SetErrorMode (SEM_NOALIGNMENTFAULTEXCEPT)*. A child process inherits its parent's error mode, so any processes created by your application after enabling this mode will also suffer the performance penalty for misaligned data. Under Windows NT for PowerPC and MIPS processors, OS-based misalignment support is disabled by default. The Alpha version of NT enables OS-based misalignment support by default; Alpha NT applications must turn this feature off if programmatic solutions are used. Setting the SEM_NOALIGNMENT-FAULTEXCEPT flag has no effect on x86 processors.

Programmatic Solutions

The first programmatic solution to data misalignment is the *__unaligned* pointer type qualifier. When the compiler sees a pointer reference (this qualifier works only with pointers) declared using the *__unaligned* type qualifier, it includes sufficient code to ensure that the memory access does not generate an alignment exception.

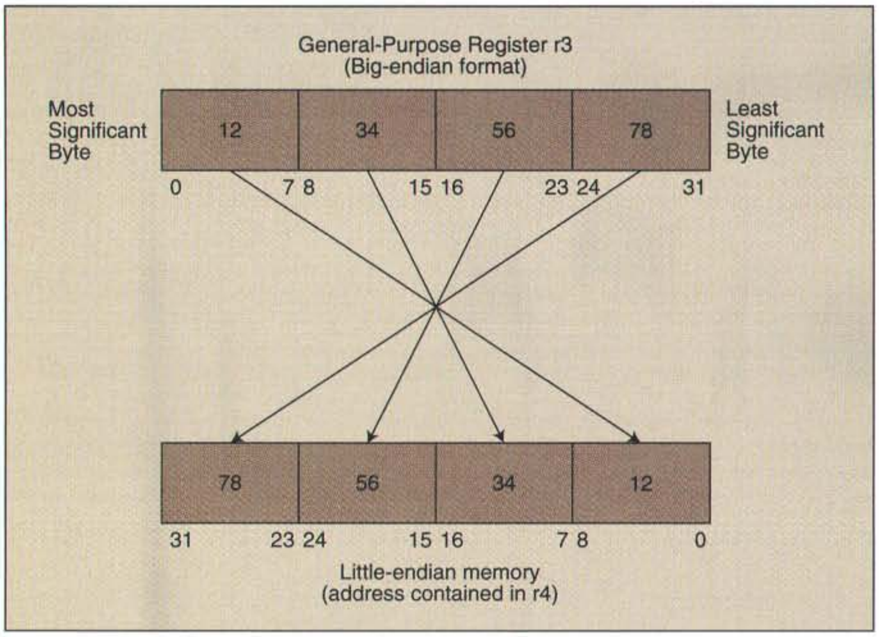


Figure 1: *Multi-byte store into Little-endian memory.*

```
#define rULONG(x)  (ULONG)(                          \
              *(UCHAR *)(&x) |                       \
              (*((UCHAR *)(&x)+1) << 8) |            \
              (*((UCHAR *)(&x)+2) << 16) |           \
              (*((UCHAR *)(&x)+3) << 24) )

#define rUSHORT(x)  (USHORT)(                        \
              *(UCHAR *)(&x) |                       \
              (*((UCHAR *)(&x)+1) << 8))
```

Figure 2: *Macros to break an access into bytes.*

```
-0 Use ONLY aligned accesses.
-1 NO alignment fix ups (causes an exception).
-2 Use OS-based fix ups for misaligned accesses.
-3 Use __UNALIGNED type qualifier.
-4 Use #PRAGMA PACK(1) directive.
```

Figure 3: *Options for the second parameter to the ALIGN program.*

In particular, it breaks multiple byte accesses into individual byte accesses. A single-alignment constrained, 32-bit load or store instruction is replaced by seven instructions that perform the same operation using only byte accesses. Similarly, a single 16-bit memory reference would be replaced by three instructions. Listing Two shows a single, 32-bit, PowerPC store-word instruction. If the store-word instructions in Listings One and Two were performed using an *__unaligned* pointer qualifier reference, the instruction would be converted to the seven instructions shown in Listing Three.

Listing Three represents compiler-generated code and, taken out of the context of the original flow of code, may appear suboptimal. Figure 1 depicts the operation of the seven instructions of Listing Three. The first instruction stores the low-order byte (0x78) of *r3* into the address contained in *r4*. The *rlwinm* instruction is used to rotate the bytes within *r3* such that each subsequent store-byte instruction references the proper value. The value contained in *r3* is in Big-endian format, and *r4* points to Little-endian memory. Therefore, the bytes must be swapped into Little-endian format during the store operation.

Another programmatic solution, the *#pragma pack()* directive, is particularly appropriate for porting between the various Windows NT platforms. One potentially recurring problem results from data structures and formats that were never designed from a portability perspective. In particular, graphics formats such as BMP and DIB do not address natural-boundary alignment. This is understandable: When these file formats were created for 80x86 software (such as Microsoft Windows), alignment issues were not a big concern—alignment was nice, but misaligned accesses didn't kill performance. With the advent of RISC processors, fixed-length instruction size, and the associated alignment restrictions, data misalignment has become an important issue.

The *#pragma pack()* directive tells the compiler two things. First, pack structure elements as close together as possible. In particular, avoid using the standard (aligned) structure padding. Second, the compiler knows to generate additional code to support misaligned accesses for elements within the "packed" structure, much like the effect of the *__unaligned* qualifier. In Listing One, a standard BMP header is packed, and the misaligned access is performed using the 32-bit *BMPDataOffset* element. This addresses application portability concerns because it allows the programmer to guarantee that the offsets within a native PowerPC structure will exactly match those defined in a native 80x86 structure.

Finally, there may be well-defined times when you know that you're about to perform a misaligned access. Instead of permanently packing a structure or declaring an *__unaligned* pointer variable to reference the memory, you can use the macros shown in Figure 2 to break the particular access into its byte components. To use the macros, simply bracket each misaligned memory reference with the appropriate macro.

The (Mis)alignment Demonstration Program

Three categories of misalignment resolution are demonstrated in the alignment program (ALIGN.C) shown in Listing One. This program lets you compare the performance impact of misaligned data by timing both aligned and misaligned accesses.

When timing a fixed number of operations in a preemptive, multitasking operating system, such as Windows NT, it is important to minimize noise in your timing measurements. To do so, elevate your thread to the highest priority possible and take multiple data samples. Listing One sets the thread-timing priority to THREAD_PRIORITY_TIME_CRITICAL. This increases the accuracy of the misalignment timing by reducing the number of interrupts that can influence the overall time required to complete the set of operations. In fact, when running ALIGN, the usefulness of your mouse will decrease dramatically.

To obtain the timing values, the *GetTimeStamp()* routine simply uses *QueryPerformanceFrequency()* and *QueryPerformanceCounter()* to sample the Win32 high-frequency counter. The time reported by ALIGN is derived by taking the difference between the time stamp before and after each set of memory-access operations.

ALIGN requires two parameters: an iteration count and one of the parameters in Figure 3. For example, the staggering 28-second measurement was generated using ALIGN 5000000 –2, which performed five million misaligned accesses using the OS to handle the alignment fixups. Listing One was used to generate the timing values shown in Table 2, which clearly demonstrates the cost of data misalignment.

Conclusion

While the increased availability of performance analysis tools and continual advancement of compiler optimization techniques will accelerate the tuning process, many key performance issues will remain embedded within the design and implementation of an application. An increased awareness among PowerPC developers of the interactions between the operating system and the microprocessor is critical to avoiding performance losses due to misaligned memory references, poor utilization of structured exception handling, and inefficient application of compiler optimizations.

For Little-endian-system implementations such as Windows NT, the means by which alignment issues are resolved can dominate application performance, as demonstrated in the ALIGN example of Listing One. Most importantly, many of the concepts in this article affect performance not only for PowerPC systems, but for other architectures, as well.

Acknowledgments

Thanks to Ray Essick and the Motorola RISC Software compiler team for a wonderfully mature PowerPC compiler.

Bibliography

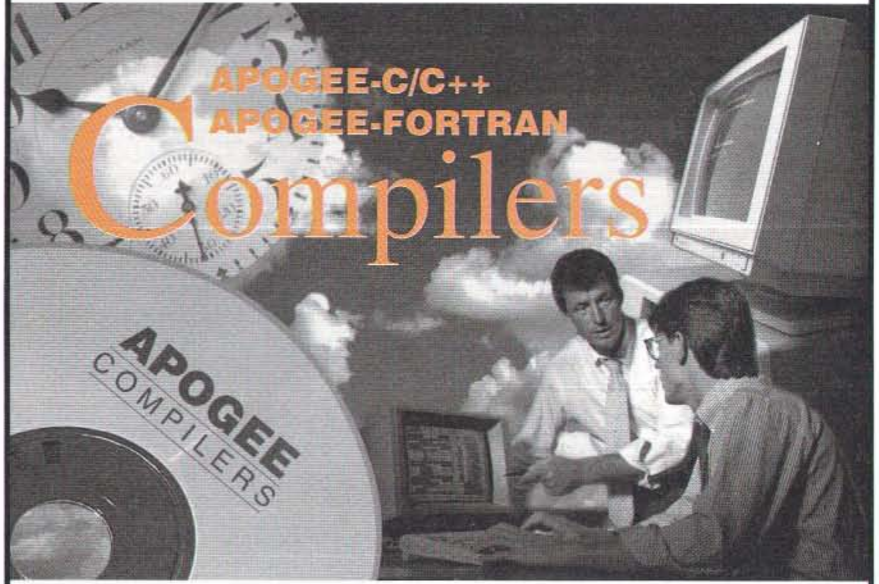
McClanahan, K. *PowerPC Programming for Intel Programmers.* San Mateo, CA: IDG Books, 1995.

Win32 Programmer's Reference, MSDN CD-ROM, July 1995.

DDJ

(Listings begin on page 38.)

When You Say Run, We Say How Fast?



- Apogee's top-rated compilers now available on **PowerPC** processors.
- Native compilers offered for both AIX **and Windows NT**.
- C/C++ compiler accepts **Microsoft's C/C++ extensions**.
- Licensed by **Microtec Research** for PowerPC embedded computer systems.
- Professional, responsive, toll free support.

APOGEE Software Inc. ™

The Compiler People

For More Information And Evaluation Copy Please Call Us At:

(800) 854-6705 info@apogee.com

Listing One

```
/*---------------------------------------------------------------------------+
 | Windows NT for PowerPC Alignment Demonstration Program                    |
 |                                                                           |
 | Mark VandenBrink, markv@risc.sps.mot.com                                  |
 | Kip McClanahan,   kip_mcclanahan@risc.sps.mot.com  -or- kip@io.com        |
 | Mike Phillip,     phillip@risc.sps.mot.com                                |
 |                                                                           |
 |                                                                           |
 +---------------------------------------------------------------------------*/
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include <windows.h>
#include <winioctl.h>
#include <string.h>
#include <ctype.h>
#include <memory.h>

// force compiler fix-ups for data accesses within the
// BMP structure by using #pragma pack(1).

#pragma pack(1)

// Standard Windows3.x BMP file header format
//
typedef struct BMPHeader {
        USHORT  FileType;                   // offset 0
        ULONG   FileSize;                   // offset 2
        USHORT  reserved1;                  // offset 6
        USHORT  reserved2;                  // offset 8
        ULONG   BMPDataOffset;              // offset 10
};

struct BMPHeader bmpBuffer;                 // declare structure

//
// Print an error message to to the screen and exit.
//
static VOID
Die(char *format, ...)
{
    va_list va;

    va_start(va, format);
    fprintf(stderr, "\n\nALIGN: ");
    vfprintf(stderr, format, va);
    ExitProcess(2);
}
```

THE ADVERTISER INDEX

RS#	Advertiser Name	Page #
27	Apogee Software	37
32	Apple Computer	C2-P1
33	Apple Computer	9
*	Apple Computer	11
*	*DDJ* Sourcebook Series	35
*	*DDJ* CD-ROM Library	48-49
*	Eastman Kodak	58
24	IBM Corp	31,32,33
31	Language Systems Corp.	27
	Matrix Resources	27
29	Metrowerks, Inc.	28
26	Motorola	C4
25	North Coast Software, Inc	2
28	Quasar Knowledge Systems	C3
23	Summit Software	10
17	Symantec	6
30	Tenon Intersystems	5

The index on this page is provided as a service to our readers. The publisher does not assume any liability for errors or omissions.
*The advertiser prefers to be contacted by phone; consult ad.

Advertising Sales Offices

Software Careers

Stephanie Dale	National Sales Manager	(415) 655-4197
Mark London	Eastern Region	(415) 655-4192
Michael Ayer	Western Region	(415) 655-4241

```
//
// Return a timestamp from the high frequency performance counters (if
// one exists).  Return the stamp in units of number of milliseconds
//
static UINT
GetTimeStamp(VOID)
{
    static DWORD FreqInMs = 0;
    LARGE_INTEGER Time;

    if (!FreqInMs) {
        if (QueryPerformanceFrequency(&Time) == TRUE) {
            if (Time.HighPart) {
                Die("Timer has too high a resolution\n");
            }
            //
            // 100-nanosecond units
            //
            FreqInMs = Time.LowPart / 1000;
        } else {
            Die("Could not get frequency of perfomance counter\n");
        }
    }
    if (QueryPerformanceCounter(&Time) == FALSE) {
        Die("System does not support high-resolution timer\n");
    }
    return Time.LowPart / FreqInMs;
}

//
// Essentially useless function that returns a value to place at
// IntPointer.  Function used to prevent compiler from optimizing
// away references to *IntPointer inside a loop.
//
DWORD
GetNextValue(VOID)
{
    static DWORD NextValue = 0;

    return NextValue++;
}

main(int argc, char **argv)
{
    CHAR Buffer[1024];
    UINT EndTime;
    UINT Max = 0;
    UINT StartTime;
    UINT i;
    struct BMPHeader *headerPtr;
    int *IntPointer2;
    __unaligned int *IntPointer;

    switch (argc) {
        case 3:
            //
            // Note: setting thread's priority to THREAD_PRIORITY_TIME_CRITICAL
            // can effectively bring a machine to its knees, depending on the
            // process priority class.
            //
            SetThreadPriority(GetCurrentThread(),
                              THREAD_PRIORITY_TIME_CRITICAL
                              );

            Max = strtoul(argv[1], 0, 0);
            //
            // The naturally aligned case
            //
            if (argv[2][0] == '-' && argv[2][1] == '0') {
                printf("ONLY aligned references\n");

                IntPointer2 = (int *)(&Buffer[4]);
                printf("Buffer at %x, IntPointer = %x\n", Buffer, IntPointer2);
                StartTime = GetTimeStamp();
                for (i = 0; i < Max; i++) {
                    *IntPointer2 = GetNextValue();
                }
                EndTime = GetTimeStamp();
                break;
            }
            //
            // The no fix-ups, alignment exception causing case
            //
            if (argv[2][0] == '-' && argv[2][1] == '1') {
                printf("NO support for misaligned references\n");

                IntPointer2 = (int *)(&Buffer[3]);
                printf("Buffer at %x, IntPointer = %x\n", Buffer, IntPointer2);
                StartTime = GetTimeStamp();
                for (i = 0; i < Max; i++) {
                    *IntPointer2 = GetNextValue();
                }
                EndTime = GetTimeStamp();
                break;
            }
            //
            // OS-based fix-ups
            //
            if (argv[2][0] == '-' && argv[2][1] == '2') {
                printf("OS support of misaligned references.\n");

                SetErrorMode(SEM_NOALIGNMENTFAULTEXCEPT);
                IntPointer2 = (int *)(&Buffer[3]);

                printf("Buffer at %x, IntPointer = %x\n", Buffer, IntPointer2);
                StartTime = GetTimeStamp();
                for (i = 0; i < Max; i++) {
                    *IntPointer2 = GetNextValue();
                }
                EndTime = GetTimeStamp();
                break;
            }
            if (argv[2][0] == '-' && argv[2][1] == '3') {
                printf("Using __UNALIGNED qualifier.\n");

                IntPointer = (int *)(&Buffer[3]);
                printf("Buffer at %x, IntPointer = %x\n",
                        Buffer,
                        IntPointer);
                StartTime = GetTimeStamp();
                for (i = 0; i < Max; i++) {
                    *IntPointer = GetNextValue();
                }
                EndTime = GetTimeStamp();
                break;
            }
            if (argv[2][0] == '-' && argv[2][1] == '4') {
                headerPtr = (struct BMPHeader *)Buffer;
                printf("Using #pragma pack(1) directive\n");

             printf("Access offset @%x\n", (ULONG)&(headerPtr->BMPDataOffset));
                StartTime = GetTimeStamp();
                for (i = 0; i < Max; i++) {
                    headerPtr->BMPDataOffset = GetNextValue();
                }
                EndTime = GetTimeStamp();

                break;
            }
      //
      // fall through
      //
      default:
       fprintf(stderr, "Usage: ALIGN number-of-iterations [-option]\n");
       fprintf(stderr, "\nwhere option is one of the following:\n");
       fprintf(stderr, "\t-0 Use ONLY aligned accesses.\n");
       fprintf(stderr, "\t-1 NO alignment fix ups (causes an exception).\n");
       fprintf(stderr, "\t-2 Use OS-based fix ups for misaligned accesses.\n");
       fprintf(stderr, "\t-3 Use __UNALIGNED type qualifier.\n");
       fprintf(stderr, "\t-4 Use #PRAGMA PACK(1) directive.\n");
       ExitProcess(0);
    }
    printf("%d milliseconds\n", EndTime - StartTime);
    ExitProcess(0);
}
```

Listing Two

```
;
; Typical 32-bit store instruction
;
; Assumes:
; r3 contains word to store at address contained in r4
;
        stw         r3, 0(r4)               ; store word contained in r3
                                            ; to address contained in r4 + 0
```

Listing Three

```
;
; The equivalent 32-bit store resulting from use of the
; __unaligned type qualifier in the pointer declaration
; for IntPointer.
; Assumes:
; r3 contains word to store at address contained in r4
;    For the purposes of this example, assume that
;    r3 = 0x12345678.
;
        stb         r3, 0(r4)               ; store the lower byte (0x78)
                                            ; of r3 into address contained
                                            ; in r4 + 0.
        rlwinm      r5, r3,24,8,31          ; extract bits 16-23 into the
                                            ; low-order position of r5
                  ; How the rlwinm instruction works:
                  ; Step 1: rotate contents of r3 left by 24 bits
                  ;         Result: 0x78123456
                  ; Step 2: generate a mask with 1-bits from
                  ;         bit 8 to 31 Result: 0x00ffffff
                  ; Step 3: AND the contents of r3 with mask and
                  ;       place the result into r5.
                  ;         Result: r5 = 0x00123456
                  ; NOTE: the next stb instruction will store
                  ;       0x56 into the address (r4 + 1).
                  ;       See Figure 1.
        stb         r5, 1(r4)        ; store next byte at r4 + 1
        rlwinm      r5, r3,16,16,31  ; extract bits 8-15 into r5
        stb         r5, 2(r4)        ; store next byte at r4 + 2
        rlwinm      r3, r3,8,24,31   ; extract bits 0-7 into r3
        stb         r3, 3(r4)        ; store final byte at r4 + 3
```

End Listings

Bit Operations with C Macros

And Knuth's MMIX as a bonus!

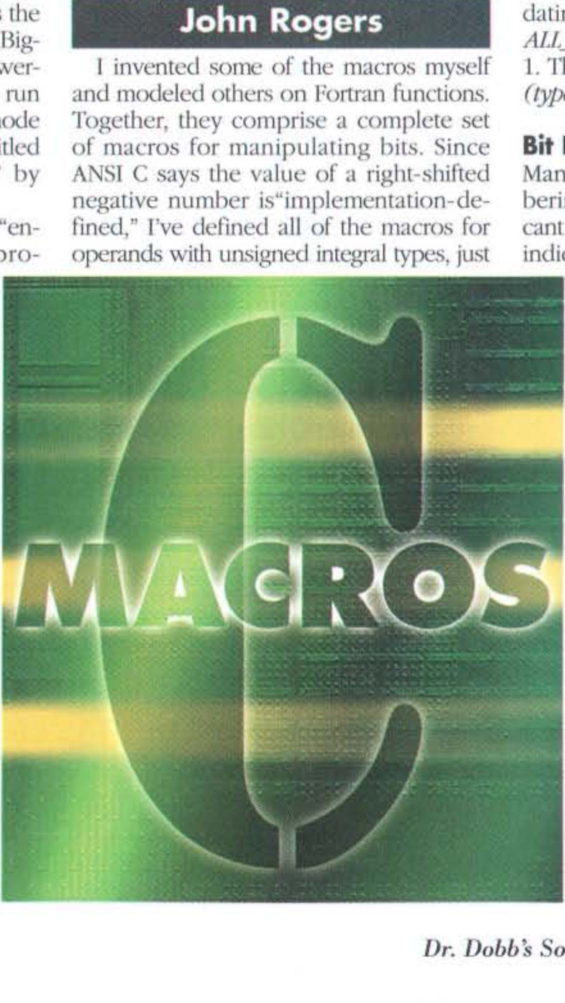
John Rogers

Endian refers to a processor addressing model that defines the byte ordering of data and instructions stored in computer memory. The most common addressing models are Big-endian (left-to-right order) and Little-endian (right-to-left). Intel-based processors (80x86, Pentium, and the like) are Little-endian, while others (such as the Motorola 680x0 in the Macintosh) are Big-endian. Still others, particularly the PowerPC, are "Bi-endian," allowing them to run in either Big-endian or Little-endian mode (see the accompanying text box entitled "PowerPC Bi-Endian Capabilities," by James R. Gillig).

As straightforward as this sounds, "endianness" can be confusing for programmers—particularly when developing portable software running on a variety of platforms. To address this confusion, I've developed an "endian engine" which handles every byte order. This engine is presented in my article "Your Own Endian Engine," (*Dr. Dobb's Journal*, November 1995). The heart of the engine is a powerful set of C macros that perform bitwise operations, which I'll discuss in this article. It's noteworthy that the examples I implement to handle these C macros are designed to handle instructions for MMIX, a hypothetical computer developed by Donald Knuth (see the accompanying text box entitled "MMIX: Knuth's New Computer").

John is a programmer in the Seattle area. He can be contacted on CompuServe at 72634,2402.

I invented some of the macros myself and modeled others on Fortran functions. Together, they comprise a complete set of macros for manipulating bits. Since ANSI C says the value of a right-shifted negative number is"implementation-defined," I've defined all of the macros for operands with unsigned integral types, just to be on the safe side. Listing One, bitops.h, has all of the C macros discussed in this article; listings begin on page 44. The complete source code to accompany this article is available electronically; see "Availability," page 3.



Except for MVBITS, all of the macros in bitops.h return values rather than updating parameters; see, for instance, the *ALL_ZERO_BITS(type)* macro in Example 1. The companion macro, *ALL_ONE_BITS (type)*, is analogous.

Bit Numbering

Many of the macros here use bit numbering. By convention, the least significant bit (LSB) is bit 0. Some of the macros indicate one or more contiguous bits in a value, using the convention of a start-bit number and a length in bits. You give the bit number of the lowest bit in the range of bits that you want. For instance, to use bits 0 through 3, give a start-bit number of 0 and a length of 4.

Conveniently, a variety of specifications and standards adhere to the LSB convention of numbering as bit 0. Most Intel processors, the IEEE MUFOM (microprocessor universal format for object modules), and the MIL-STD FORTRAN functions all use this convention. The only major exception is IBM mainframes, which number the most significant bit (MSB) as bit 0.

Fortran-Inspired Macros

Since at least the 1970s, many versions of Fortran have included a standard set of bit functions that include the usual operations: AND, OR, NOT, and exclusive-OR. These Fortran functions also include routines for bit extraction, insertion, shift, and circular shift. Rather than reinvent the wheel, I've used the same function names and operand orders. However, since the Fortran functions were designed for implementations with just one integer type, and C has many sizes of integer types (ranging from *char* to *long*), I added, where necessary, an additional parameter (at the end) to indicate the data type to be returned. This must be some unsigned integral type.

As for the Fortran-inspired macros, I'll start with *NOT(value,type) (bitwise complement)*, sometimes called the "flip bits" or "invert" operation. In Fortran, the *NOT(value)* function has one operand (an integer value) and returns an integer: the inverted value. The C *NOT(value,type)* macro has an additional operand, which must be an unsigned integral data type. The *NOT(value,type)* macro converts the given value to the given type and returns the converted value with all of the bits inverted. For instance, in an implementation with 8-bit characters, *NOT(0xF0, unsigned char)* would be 0x0F.

The *IAND(m,n,type)* ("integer and") macro simply performs a bitwise-AND of the bits in the first two operands, which are treated as type *type*, and returns the result. It supports any unsigned integer type for its operands. Kenneth Hamilton used the Fortran version of this macro in his article "Direct Memory Access from PC Fortrans" (*Dr. Dobb's Journal,* May 1993). Hamilton's code needed to extract the low byte from some integer value. Using the C macros in bitops.h, the equivalent would be:

```
unsigned int ic1;
ic1=IAND(ic,255,unsigned int);
```

There is an integer extract bits (IBITS) macro that extracts and right-justifies one or more contiguous bits from a given value. IBITS is called as *IBITS(value, bitnum,len,type)*. For instance, IBITS *(0x5678,8,4,unsigned long)* returns an unsigned-long value of 0x6.

Another Fortran-inspired macro is the integer shift (ISHFT) macro, a call to which appears as *ISHFT(value,shifts,type)*. ISHFT and its circular-shift companion ISHFTC are unique in that they indicate the direction to shift by positive or negative values of the *shifts* parameter. A positive value for *shifts* causes a left shift by that many bits; a negative value causes a right shift by that many bits; a 0 value causes no shift. Make sure the absolute value of *shifts* is less than or equal to the size of type in bits; otherwise, the result is undefined.

For a Fortran version of the *ishft* routine, see Ray Duncan's "16-Bit Software Toolbox" column (*Dr. Dobb's Journal,* August 1985).

Unlike the other macros in bitops.h, MVBITS updates a value (using a pointer passed to it) rather than returning a value. The call *MVBITS(src,srcindex,len,destptr,destindex,type)* updates the value at **destptr* (starting at bit *destindex* for *len* bits) with *len* bits extracted from *src* starting at bit number *srcindex*. Example 3 shows an example of using MVBITS.

BitOps Examples Using MMIX

In the examples from here on, I'll use the bitops.h C macros to handle instructions for MMIX. The parts of an MMIX instruction are multiples of 8 bits each, but the bitops.h macros don't depend on this.

A normal MMIX instruction is 32 bits long and broken into four fields of 8 bits each; see Table 1. Three fields generally refer to registers or contain immediate values. Knuth refers to these fields as X, Y, and Z. In some other instructions, Knuth

Contents	Start-bit #	Len
opcode	24	8
X (usually target register)	16	8
Y (usually source register)	8	8
Z (usually source register)	0	8

Table 1: *MMIX normal instruction layout.*

```
unsigned short x;
x = ALL_ZERO_BITS(unsigned short);
```

Example 1: *Simple C macro.*

```
#include "mmixcom.h"  /* MMIX_Opcode_T, etc. */

MMIX_Instr_T  Current_Instruction;
MMIX_Opcode_T Current_Opcode;
   ...
/* Assume Current_Instruction has already been set. */
/* IBITS right-justifies result, so use it to extract opcode. */
Current_Opcode = (MMIX_Opcode_T) IBITS(
      Current_Instruction,          /* value */
      MMIX_INSTR_OPCODE_START,      /* start bit num */
      MMIX_INSTR_OPCODE_LEN,        /* len */
      MMIX_Instr_T);                 /* type */
```

Example 2: *Extracting an opcode using the IBITS macro.*

```
MMIX_Instr_T  New_Instr = ALL_ZERO_BITS(MMIX_Instr_T);

/* Set opcode. */
MVBITS(
      0xC2,                      /* ADDU opcode */   /* src */
      0,                         /* src index: src bit 0. */
      MMIX_INSTR_OPCODE_LEN,    /* len */
      &New_Instr,                /* dest ptr */
      MMIX_INSTR_OPCODE_START,  /* dest index */
      MMIX_Instr_T);             /* type */

/* Set X (target) field to say r40. */
MVBITS(
      40,                        /* register 40 */  /* src */
      0,                         /* src index: src bit 0. */
      MMIX_INSTR_X_LEN,         /* len */
      &New_Instr,                /* dest ptr */
      MMIX_INSTR_X_START,       /* dest index */
      MMIX_Instr_T);             /* type */

/* Set Y (a source field) to r41. */
MVBITS(
      41,                        /* register 41 */  /* src */
      0,                         /* src index: src bit 0. */
      MMIX_INSTR_Y_LEN,         /* len */
      &New_Instr,                /* dest ptr */
      MMIX_INSTR_Y_START,       /* dest index */
      MMIX_Instr_T);             /* type */

/* Set Z (the other source field) to r42. */
MVBITS(
      42,                        /* register 42 */  /* src */
      0,                         /* src index: src bit 0. */
      MMIX_INSTR_Z_LEN,         /* len */
      &New_Instr,                /* dest ptr */
      MMIX_INSTR_Z_START,       /* dest index */
      MMIX_Instr_T);             /* type */
```

Example 3: *Creating an instruction with the MVBITS macro.*

```
#include "mmixcom.h"  /* MMIX_Word_T, MMIX_WORD_LEN, etc. */

MMIX_Word_T
Sim_SRU(  /* Simulate shift right unsigned instr. */
   MMIX_Word_T  Source_Reg,
   MMIX_Word_T  Shift_Count_Reg)
{
   if (Shift_Count_Reg >= MMIX_WORD_LEN)
      return (0);
   return (RIGHT_SHIFT_BITS(
         Source_Reg,              /* value */
         Shift_Count_Reg,         /* shifts */
         MMIX_WORD_LEN,           /* len */
         MMIX_Word_T));           /* type */
}
```

Example 4: *Using the RIGHT_SHIFT_BITS macro to simulate an SRU instruction.*

```
MMIX_Word_T
Sim_XOR(  /* Simulate exclusive-OR bits instr. */
   MMIX_Word_T  Some_Bits,
   MMIX_Word_T  Other_Bits)
{
   return (IEOR(
         Some_Bits,
         Other_Bits,
         MMIX_Word_T));  /* type */
}
```

Example 5: *Using the IEOR macro to simulate an XOR instruction.*

combines the Y and Z fields for 16 bits. In still other instructions, he combines the X, Y, and Z fields into a 24-bit field.

For starters, I'll define a type to hold an instruction, keeping in mind that ANSI C implicitly requires *unsigned long* to hold 32 bits or more. Remember that ANSI C does not require any particular byte order when storing larger-than-byte objects in memory. Using a trailing *_T* convention to indicate a type, you can define a type *(MMIX_Instr_T)* to contain the object code for one instruction, as shown in mmixcom.h (Listing Two).

Taking advantage of the implicit ANSI C requirement that *unsigned char* be 8 bits or larger, mmixcom.h also defines types for bytes in general and instruction opcodes in particular. I call these types *MMIX_Byte_T* and *MMIX_Opcode_T*, respectively.

To use the bitops.h macros to extract the opcode from an instruction, for example, you need to specify start-bit numbers and bit lengths. Listing Two also contains equates called MMIX_INSTR_OPCODE_START and MMIX_INSTR_OPCODE_LEN for this.

Given those bit numbers, you can use the IBITS (integer extract bits) macro to extract the opcode from an instruction; see Example 2. You will recall that IBITS right-justifies its result.

MMIX also stores the X, Y, and Z fields as bytes. Example 3 shows how to create an instruction from scratch using MVBITS. In this case, I am creating an instruction to set *r40* (register 40) to the unsigned sum of registers 41 and 42.

Shifting Bits with MMIX and bitops.h

MMIX has an SRU (shift right unsigned) instruction. *SRU r3=r4>>r5* is a shift-right unsigned instruction in Knuth's current assembler syntax that sets register 3 to register 4 shifted right by the number of bits indicated in register 5. If the value in register 5 is greater than or equal to the size of a register in bits, then register 3 will be set to zero. Example 4 shows a short routine that simulates the SRU instruction using the bitops.h RIGHT_SHIFT_BITS macro.

You can readily emulate MMIX's XOR

MMIX: Knuth's New Computer

Back in the 1960s, Donald Knuth designed a hypothetical computer called "MIX" for his *Art of Computer Programming* algorithms books. MIX shows its age in various ways, so Knuth is designing a RISC-like successor called "MMIX" (short for "Meta-MIX" or "Mega-MIX"). He started from scratch to avoid the restrictions of the old architecture. The new computer incorporates Big-endian byte ordering, byte addressing, two's-complement integer arithmetic, and IEEE floating-point arithmetic.

Knuth has not yet published his description of MMIX. His latest draft is dated August 20, 1992. He expects to make many technical changes in his next draft, due sometime in 1995, so details given here may change as well.

Knuth plans to use MMIX for the later volumes of the *Art of Computer Programming* series. I myself hope to write a cross assembler and simulator for MMIX for publication in *Dr. Dobb's Journal*. This drives my exploration of "big-integer" (64-bits or more) routines for C, as well as 64-bit, portable object-file formats (like MUFOM and ELF).

In MMIX, Knuth has adopted the common definition of a byte as having 8 bits. He is much more generous with registers; MMIX has 256 general-purpose registers. Knuth has also accounted for other practical issues this time. His description of MMIX floating point acknowledges that on some models, the system may trap floating-point "instructions" and interpret them in software. MMIX is supposed to have virtual memory, although the current draft doesn't seem to have enough detail for an operating system to deal with page faults or page tables.

The 1992 draft defines a 32-bit system, but Knuth is likely to convert to 64 bits before he publishes the final version. He has also had second thoughts about a number of complications the draft introduced: regions, probable branches, and delay slots, for example.

Regions are a simple way to provide multiple address spaces, kind of like segment registers. The delay slot avoids having to refill the prefetch buffer because of the branch. I first saw delay slots being used on the MIPS when I worked at Microsoft. Many of us with previous assembly-language experience kept forgetting that the instruction in the delay slot would execute, too.

Probable branches and delay slots are, in my opinion, architectural warts to improve pipeline performance while driving the assembly-language programmer crazy. In a pipelined system, the instruction right after a branch has already been prefetched, so why not execute it?

It seems, however, that Knuth is reconsidering these additions, and they will probably be dropped from the next draft.

—J.R.

instruction using the bitops.h IEOR (integer exclusive-OR) macro; see Example 5.

MMIX is perhaps unique among instruction sets in having a nor bits (NOR) instruction. You may be familiar with NOR and NAND gates from digital electronics. The corresponding bitops.h macro is NOR_BITS. Example 6 shows how the NOR_BITS macro may be used to simulate MMIX's NOR instruction.

```
MMIX_Word_T
Sim_NOR(  /* Simulate NOR bits instr. */
   MMIX_Word_T  Some_Bits,
   MMIX_Word_T  Other_Bits)
{
   return (NOR_BITS(
          Some_Bits,
          Other_Bits,
          MMIX_Word_T));  /* type */
}
```

Example 6: *Using the NOR_BITS macro to simulate a NOR instruction.*

Conclusion

C provides some powerful bitwise operators, although you need to be careful regarding the widening of values between different data types. The macros in bitops.h provide a more complete set of bitwise operations than bare C, with protection against widening problems, although you must avoid macro arguments with side effects. You should also avoid using any of the signed data types with the macros.

References

ANSI X3.159-1989, American National Standard for Information Systems—Programming Language—C. New York, NY: American National Standards Institute (ANSI), 1989.

Knuth, Donald E. MMIX. Private communication, August 20, 1992.

MIL-STD-1753. *Military Standard: FORTRAN, DOD Supplement to American National Standard X3.9-1978.* November 9, 1978. Available free from the Defense Printing Service at 215-697-2179.

DDJ

(Listings begin on page 44.)

PowerPC Bi-Endian Capabilities

Jim Gillig

The PowerPC is a Bi-endian RISC processor that supports both Big- and Little-endian addressing models. The Bi-endian architecture provides hardware and software developers with the flexibility to choose either mode when migrating operating systems and applications from their current BE or LE platforms to the PowerPC. Program instructions are like multibyte-scalar data and are subject to the byte-order effect of Endianness.

Each individual PowerPC machine instruction occupies an aligned word in storage as a 32-bit integer containing that instruction's value. In general, the appearance of instructions in memory is of no concern to the programmer. Program code in memory is inherently either a LE or BE sequence of instructions even if it is an Endian-neutral implementation of an algorithm.

How does the PowerPC handle both LE and BE addressing models? The processor calculates the effective address of data and instructions in the same manner whether in BE mode or LE mode; when in LE mode only, the PowerPC implementation further modifies the effective address to provide the appearance of LE memory to the program for loads and stores.

The operating system is responsible for establishing the Endian mode in which processes execute. Once a mode is selected, all subsequent memory loads and stores will be affected by the memory-addressing model defined for that mode. Byte-alignment and performance issues need to be understood before using an endian mode for a given application. Alignment interrupts may occur in LE mode for the following load and store instructions:

- Fixed-point load instructions.
- Fixed-point store instructions.
- Load-and-store with byte-reversal instructions.
- Fixed-point load-and-store multiple instructions.
- Fixed-point move-assist instructions.
- Storage-synchronization instructions.
- Floating-point load Instructions.
- Floating-point store instructions.

For multibyte-scalar operations, when executing in LE mode, the current PowerPC processors take an alignment interrupt whenever a load or store instruction is issued with a misaligned effective address, regardless of whether such an access could be handled without causing an interrupt in BE mode. For code that is compiled to execute on the PowerPC in LE mode, the compiler should generate as much aligned data and as many aligned instructions as possible to minimize the alignment interrupts. Generally, more alignment interrupts will occur in LE mode than in BE mode. When an alignment interrupt occurs, the operating system should handle the interrupt by software emulation of the *load* or *store*.

A very powerful feature of the PowerPC architecture is the set of integer load-and-store instructions with byte reversal that allow applications to interchange or convert data from one Endian type to the other, without performance penalty. These load-and-store instructions are *lhbrx/sthbrx*, load/store halfword byte-reverse indexed and *lwbrx/stwbrx*, load/store word byte-reverse indexed. They are ideal for emulation programs that handle LE-type instructions and data, such as the emulation of the Intel instruction set and data. These instructions significantly improve performance in loading and storing LE data while executing PowerPC instructions in BE mode and emulating the Intel instruction behavior; this eliminates the byte-alignment and data-conversion overhead found in architectures that lack byte-reversal instructions. Currently, these instructions can be accessed only through assembly language. Until C compilers provide support to automatically generate the right load and store instructions for this type of data, C programs can rely on masking and concatenating operations or embed the assembly-language byte-reversal instructions.

James R. is a software engineer on OS/2 and IBM Workplace technologies in Boca Raton, FL. He can be reached through the DDJ *offices.*

Listing One

```
/* BitOps.h - bit operation macros. Copyright (c) 1987-1994 by JR
 * (John Rogers). All rights reserved. CompuServe: 72634,2402
 * Permission is granted to use these macros in compiled code without payment
 * of royalties or inclusion of a copyright notice.  This source file
 * may not be sold without written permission from the author.
 *
 * The following macros are inspired by the FORTRAN bit operation routines in
 * MIL-STD-1753. Except for MVBITS, all return values rather than modifying
 * parameters. MVBITS updates a parameter and does not return anything. The
 * leading "I" in most of these names means that they return some kind of
 * integer result.
 *    BTEST(value,bitnum,type)
 *    IAND(m,n,type)
 *    IBCLR(value,bitnum,type)
 *    IBITS(value,bitnum,len,type)
 *    IBSET(value,bitnum,type)
 *    IEOR(m,n,type)
 *    IOR(m,n,type)
 *    ISHFT(value,shifts,type)
 *    ISHFTC(value,shifts,len,type)
 *    MVBITS(src,srcindex,len,destptr,destindex,type)
 *    NOT(value,type)
 * The following C macros were invented by me (JR) or various other C
 * programmers; all return values rather than modifying parameters:
 *    ALL_ONE_BITS(type)
 *    ALL_ZERO_BITS(type)
 *    BIT_NUM_AND_LEN_TO_MASK(bitnum,len,type)
 *    BIT_NUM_TO_MASK(bitnum,type)
 *    CLEAR_BITS_USING_MASK(value,mask,type)
 *    FLIP_BITS_USING_MASK(value,mask,type)
 *    LEFT_CIRCULAR_SHIFT_BITS(value,shifts,len,type)
 *    LEFT_SHIFT_BITS(value,shifts,len,type)
 *    NAND_BITS(m,n,type)
 *    NOR_BITS(m,n,type)
 *    RIGHT_CIRCULAR_SHIFT_BITS(value,shifts,len,type)
 *    RIGHT_SHIFT_BITS(value,shifts,len,type)
 *    SET_BITS_USING_MASK(value,mask,type)
 *    TEST_BITS_USING_MASK(value,mask,type)
 *    TYPE_SIZE_IN_BITS(type)
 *    XNOR_BITS(m,n,type)
 * Beware of side effects: many macros in this file evaluate their arguments
 * more than once.  These are marked with EVALTWICE comments.
 */

/* Gracefully allow multiple includes of this file. */
#ifndef BITOPS_H
#define BITOPS_H

/****************** I N C L U D E S ****************/
/*lint -efile(766,limits.h) */
#include <limits.h>    /* CHAR_BIT. */

/*********************** M A C R O S  ****************/
/* ALL_ONE_BITS(type): Generate a value of type "type" with all bits set to
 * 1. "type" must be an unsigned integral type.
 */
#define ALL_ONE_BITS(type)      ( (type) ~((type)0) )

/* ALL_ZERO_BITS(type): Generate a value of type "type" with all bits set to
 * 0.  "type" must be an unsigned integral type.
 */
#define ALL_ZERO_BITS(type)     ( (type) 0 )

/* BIT_NUM_AND_LEN_TO_MASK(bitnum,len,type): Return a mask of type "type",
 * with "len" bits on, starting at "bitnum".  Bit 0 is LSB, bits start at
 * "bitnum" and are turned on in the mask starting at "bitnum" and going to
 * the left. "type" must be an unsigned integral type.
 */
/*EVALTWICE*/
#define BIT_NUM_AND_LEN_TO_MASK(bitnum,len,type) \
   /*CONSTCOND*/ \
   /*lint -save -e506 -e572 -e778 */ \
   ( (type) \
      ( \
         ( (ALL_ONE_BITS(type)) \
            >> ((TYPE_SIZE_IN_BITS(type)) \
               - ((bitnum)+(type)(len)) ) ) \
         &  ( \
               ((bitnum)>0) \
               ? ~( ALL_ONE_BITS(type) \
                  >> ( (TYPE_SIZE_IN_BITS(type)) \
                     - (bitnum) ) ) \
               : ALL_ONE_BITS(type) ) \
      ) \
   ) \
   /*lint -restore */
/* BIT_NUM_TO_MASK(bitnum,type): Convert bit number "bitnum" to mask of type
 * "type".  Bits are numbered from right to left, with bit 0 being the least
 * significant bit (LSB). "type" must be an unsigned integral type.
 * This is my (JR's) modification of something posted to Usenet by Bill
 * Shannon (shannon@sun.uucp) many years ago.
 */
#define BIT_NUM_TO_MASK(bitnum,type) \
   ( (type) ( ((type)1) << ((type)(bitnum)) ) )

/* BTEST(value,bitnum,type): Test bit numbered "bitnum" in "value", which must
 * be of type "type".  If the tested bit is on, return a Boolean true value
 * (1); otherwise, return a Boolean false (0). "type" must be an unsigned
 * integral type.
 */
#define BTEST(value,bitnum,type) \
   ( ( (value) & (BIT_NUM_TO_MASK((bitnum),type)) ) \
     ? 1 : 0 \
   )
/* CLEAR_BITS_USING_MASK(value,mask,type): Return "value", except that any bits
 * which are turned on in "mask" will be turned off in the return value.
 * "type" must be an unsigned integral type.
 */
#define CLEAR_BITS_USING_MASK(value,mask,type) \
   ( (type) ( (value) & ~(mask) ) )
/* FLIP_BITS_USING_MASK(value,mask,type): Return "value", except that any bits
 * which are turned on in "mask" will be flipped (toggled) in the return
 * value.  "type" must be an unsigned integral type.
 */
#define FLIP_BITS_USING_MASK(value,mask,type) \
   ( (type) ( (value) ^ (mask) ) )
/* IAND(m.n,type): Return the bitwise "and" of the integral values "m" and "n".
 * "type" must be an unsigned integral type.
 */
#define IAND(m,n,type) \
   ( (type) ( (m) & (n) ) )
/* IBCLR(value,bitnum,type):  Return "value" with bit at "bitnum" cleared
 * (zeroed). "type" must be an unsigned integral type.
 */
#define IBCLR(value,bitnum,type) \
   ( \
      (type) CLEAR_BITS_USING_MASK( \
         (value), \
         BIT_NUM_TO_MASK( (bitnum), type ), \
         type) \
   )
/* IBITS(value,bitnum,len,type): Extract bits from "value", starting at bit
 * "bitnum", for "len" bits. The result will be right justified. "type" must be
 * an unsigned integral type.
 */
/*EVALTWICE*/
#define IBITS(value,bitnum,len,type) \
   /*CONSTCOND*/ \
   /*lint -save */ /* Preserve PC-LINT options. */ \
   /*lint -e572 */ /* Ignore excessive shift val */ \
   /*lint -e778 */ /* Ignore const expr eval to 0 */ \
   ( (type) \
      ( \
         ( (value) & \
            (BIT_NUM_AND_LEN_TO_MASK( \
               (bitnum), (len), type )) ) \
         >> (bitnum) \
      ) \
   ) \
   /*lint -restore */
/* IBSET(value,bitnum,type):  Return "value" with bit at "bitnum" set to true.
 * "type" must be an unsigned integral type.
 */
#define IBSET(value,bitnum,type) \
   ( (type) \
      ( \
         SET_BITS_USING_MASK( \
            (value), \
            BIT_NUM_TO_MASK( (bitnum), type ), \
            type) \
      ) \
   )
/* IEOR(m.n,type): Return the bitwise exclusive-or of the integral values "m"
 * and "n".  "type" must be an unsigned integral type.
 */
#define IEOR(m,n,type) \
   ( (type) ( (m) ^ (n) ) )
/* IOR(m.n,type): Return the bitwise "or" of the integral values "m" and "n".
 * "type" must be an unsigned integral type.
 */
#define IOR(m,n,type) \
   ( (type) ( (m) | (n) ) )
/* ISHFT(value,shifts,type): Return "value" with bits logically shifted as
 * specified by "shifts".  Zeros will be shifted-in as applicable. A positive
 * amount for "shifts" causes a left shift; a negative amount causes a right
 * shift; a zero amount causes no shift. Note that the absolute value of
 * "shifts" must be less than or equal to TYPE_SIZE_IN_BITS("type"). Also note
 * that "value" must be of type "type", and "type" must be an unsigned
 * integral type.
 */
/*EVALTWICE*/
#define ISHFT(value,shifts,type) \
   /*CONSTCOND*/ \
   /*lint -save */ /* Preserve PC-LINT settings. */ \
   /*lint -e504 */ /* Ignore unusual shift value */ \
   /*lint -e778 */ /* Ignore const expr eval to 0 */ \
   ( (type) \
      ( ((shifts)>0) \
         ? ( (value) << (shifts) ) \
         : ( ( (shifts)<0 ) \
            ? ( (value) >> (-(shifts)) ) \
            : (value) \
            ) \
      ) \
   ) \
   /*lint -restore */
/* ISHFTC(value,shifts,len,type): Return "value" with bits circularly shifted
 * (as specified by "shifts") within the lower "len" bits of "value". A
 * positive amount for "shifts" causes a left shift; a negative amount causes
 * a right shift; a zero amount causes no shift. Note that the absolute value
 * of "shifts" must be less than or equal to "len".  Also note that "value"
 * must be of type "type", and "type" must be an unsigned integral type. "len"
 * must be greater than 0 and less than or equal to TYPE_SIZE_IN_BITS("type").
 */
/*EVALTWICE*/
#define ISHFTC(value,shifts,len,type) \
   /*lint -save -e501 */ \
   ( (type) ( \
      ( ((shifts) == 0) \
         || ((len) == (type) (shifts)) \
         || ((len) == - (type) (shifts)) ) \
      ? ((type)(value)) \
      : ( \
        ( (shifts) > 0 ) \
        ? (RIGHT_CIRCULAR_SHIFT_BITS( \
          (value), (shifts), (len), type) ) \
        : (LEFT_CIRCULAR_SHIFT_BITS( \
          (value), (type) (- (shifts)), \
          (len), type) ) ) \
     ) \
   ) /*lint -restore */
/* LEFT_CIRCULAR_SHIFT_BITS(value,shifts,len,type): Return "value" with bits
 * circularly shifted left "shifts" bits within the lower "len" bits of
 * "value". A zero amount for "shifts" causes no shift. Note that "shifts"
 * must be less than or equal to  "len". Also note that "value" must be of type
 * "type", and "type" must be an unsigned integral type. "len" must also be
 * greater than zero and less than or equal to TYPE_SIZE_IN_BITS("type").
 */
/*EVALTWICE*/
#define LEFT_CIRCULAR_SHIFT_BITS( \
  value,shifts,len,type) \
  /*lint -save -e5Ø4 */ \
  ( (type) ( \
     ((shifts)==Ø) ¦¦ ((len)==(type) (shifts)) \
     ? (value) \
     : ( ( (value) & \
          ~BIT_NUM_AND_LEN_TO_MASK(Ø,(len),type) ) \
       ¦ ( ( (value) & (BIT_NUM_AND_LEN_TO_MASK( \
          Ø, (len), type )) ) \
          >> (shifts) ) \
       ¦ ( ( (value) & (BIT_NUM_AND_LEN_TO_MASK( \
          Ø, (shifts), type )) ) \
          << ((len)-(type)(shifts)) ) ) ) \
   ) /*lint -restore */
/* LEFT_SHIFT_BITS(value,shifts,len,type): Return "value" with bits logically
 * shifted left "shifts" bits within the lower "len" bits of "value".  If
 * necessary, zero bits are added on the right.  A zero amount for "shifts"
 * causes no shift. Note that "shifts" must be less than or equal to "len".
 * Also note that "value" must be of type "type", and "type" must be an
 * unsigned integral type. "len" must also be greater than zero and less than
 * or equal to TYPE_SIZE_IN_BITS("type").
 */
/*EVALTWICE*/
#define LEFT_SHIFT_BITS(value,shifts,len,type) \
   /*lint -save -e5Ø4 */ \
   ( (type) ( \
      ( ((shifts)==Ø) ¦¦ ((len)==(type) (shifts)) ) \
      ? (value) \
      : ( ( (value) & \
           ~BIT_NUM_AND_LEN_TO_MASK(Ø,(len),type) ) \
        ¦ ( ( (value) << (shifts) ) \
           & (BIT_NUM_AND_LEN_TO_MASK( \
              Ø, (len), type )) ) ) \
      ) \
   ) /*lint -restore */
/* MVBITS(src,srcindex,len,destptr,destindex,type): Update the value that
 * "destptr" points to, using bits extracted from "src" starting at bit
 * "srcindex" for "len" bits.  "destindex" indicates the bit number in the
 * destination to begin updates.  "type" must be an unsigned integral type.
 */
/*EVALTWICE*/
#define MVBITS( \
   src,srcindex,len,destptr,destindex,type) \
   /*CONSTCOND*/ /*lint -save -e5Ø6 */ \
   { \
       type srcbits = \
          (src) & BIT_NUM_AND_LEN_TO_MASK( \
                 (srcindex), (len), type ); \
       type destmask = BIT_NUM_AND_LEN_TO_MASK( \
                 (destindex), (len), type ); \
       *(destptr) &= ~destmask; \
       *(destptr) ¦= ISHFT( \
                 srcbits, \
                 (int) ((destindex)-(srcindex)), \
                 type ); \
   } /*lint -restore */
/* NAND_BITS(m.n,type): Return the bitwise "nand" of the integral values
 * "m" and "n".  "type" must be an unsigned integral type.
 */
#define NAND_BITS(m,n,type) \
   ( (type) ~ ( IAND((m),(n),type) ) )
/* NOR_BITS(m.n,type): Return the bitwise "nor" of the integral values "m" and
 * "n". "type" must be an unsigned integral type.
 */
#define NOR_BITS(m,n,type) \
   ( (type) ~ ( IOR((m),(n),type) ) )
/* NOT(value,type): Return all bits of "value" flipped. Note that "value" must
 * be of type "type", which must be an unsigned integral type.
 */
#define NOT(value,type)  ( (type) ~((type)(value)) )
/* RIGHT_CIRCULAR_SHIFT_BITS(value,shifts,len,type). Return "value" with bits
 * circularly shifted right "shifts" bits within the lower "len" bits of
 * "value".  A zero amount for "shifts" causes no shift. Note that "shifts"
 * must be less than or equal to "len". Also note that "value" must be of type
 * "type", and "type" must be an unsigned integral type. "len" must also be
 * greater than zero and less than or equal to TYPE_SIZE_IN_BITS("type").
 */
/*EVALTWICE*/
#define RIGHT_CIRCULAR_SHIFT_BITS( \
   value,shifts,len,type) \
  /*lint -save -e5Ø4 */ \
  ( (type) ( \
  ((shifts)==Ø) ¦¦ ((len)==(type) (shifts)) \
  ? (value) \
  : ( ( (value) & \
        ~BIT_NUM_AND_LEN_TO_MASK(Ø,(len),type) ) \
    ¦ ( ( (value) & (BIT_NUM_AND_LEN_TO_MASK( \
      Ø, ((len)-(type)(shifts)),type)) ) \
      <<(shifts)) \
    ¦ ( ((value)&(BIT_NUM_AND_LEN_TO_MASK( \
      ((len)-(type)(shifts)),(shifts),type))) \
      >> ((len)-(type)(shifts)) ) ) ) \
   ) /*lint -restore*/
/* RIGHT_SHIFT_BITS(value,shifts,len,type): Return "value" with bits logically
 * shifted right "shifts" bits within the lower "len" bits of "value".  If
 * necessary, zero bits are added on the left. A zero amount for "shifts"
 * causes no shift. Note that "shifts" must be less than or equal to "len".
 * Also note that "value" must be of type "type", and "type" must be an
 * unsigned integral type. "len" must also be greater than zero and
 * less than or equal to TYPE_SIZE_IN_BITS("type").
 */
/*EVALTWICE*/
#define RIGHT_SHIFT_BITS(value,shifts,len,type) \
   /*lint -save -e5Ø4 */ \
   ( (type) ( \
      ( ((shifts)==Ø) ¦¦ ((len)==(type) (shifts)) ) \
      ? (value) \
      : ( ( (value) & \
           ~BIT_NUM_AND_LEN_TO_MASK(Ø,(len),type) ) \
        ¦ ( ( (value) & (BIT_NUM_AND_LEN_TO_MASK( \
           Ø, (len), type )) ) >> (shifts) ) ) \
      ) \
   ) /*lint -restore */
/* SET_BITS_USING_MASK(value,mask,type): Return "value", except that any bits
 * which are turned on in "mask" will also be turned on in the return
 * value. "type" must be an unsigned integral type.
 */
#define SET_BITS_USING_MASK(value,mask,type) \
   ( (type) ( (value) ¦ (mask) ) )
/* TEST_BITS_USING_MASK(value,mask,type): Return "value", except that only bits
 * which are turned on in "mask" will be returned. "type" must be an
 * unsigned integral type.
 */
#define TEST_BITS_USING_MASK(value,mask,type) \
   ( (type) ( (value) & (mask) ) )
/* TYPE_SIZE_IN_BITS(type): Return the number of bits required for type "type".
 */
#define TYPE_SIZE_IN_BITS(type) \
   ( (type) ( sizeof(type) * CHAR_BIT ) )
/* XNOR_BITS(m.n,type): Return the bitwise exclusive "nor" of the integral
 * values "m" and "n".  "type" must be an unsigned integral type.
 */
#define XNOR_BITS(m,n,type) \
   ( (type) ~ ( IEOR((m),(n),type) ) )

#endif /* BITOPS_H */
```

Listing Two

```
/* mmixcom.h--  MMIX common defns. Copyright (c) 1994 by JR (John Rogers).
 * All rights reserved. CompuServe: 72634,24Ø2
 * FUNCTION - mmixcom.h contains types and equates used for defining MMIX
 *      instructions in object code format.
 * We take advantage of the implicit ANSI C requirement that unsigned char be 8
 * bits or larger. Similarly, we can assume unsigned long is 32 bits or larger.
 */

#ifndef MMIXCOM_H
#define MMIXCOM_H

/* Define a type for one instruction. Note that this will be at least 32 bits,
 * depending on the compiler.
 */
typedef unsigned long  MMIX_Instr_T;

/* We also need to deal with single words in MMIX. These are currently 32 bits
 * wide, although Knuth is likely to change them to 64 bits soon.
 */
typedef unsigned long  MMIX_Word_T;
#define MMIX_WORD_LEN  32

/* Many parts of MMIX words are in bytes. In MMIX, a byte is 8 bits long. In C,
 * this might be larger.
 */
typedef unsigned char  MMIX_Byte_T;

/* Even if "char" is more than 8 bits, leave this. */
#define MMIX_BYTE_BIT_LEN  8

/* Define a type for an opcode. */
typedef MMIX_Byte_T  MMIX_Opcode_T;

/* Define equates for each part of MMIX_Instr_T. Use bit numbering convention
 * of Ø=least significant bit (LSB).
 */
#define MMIX_INSTR_OPCODE_START 24
#define MMIX_INSTR_OPCODE_LEN   MMIX_BYTE_BIT_LEN

#define MMIX_INSTR_X_START  16
#define MMIX_INSTR_X_LEN    MMIX_BYTE_BIT_LEN

#define MMIX_INSTR_Y_START  8
#define MMIX_INSTR_Y_LEN    MMIX_BYTE_BIT_LEN

#define MMIX_INSTR_Z_START  Ø
#define MMIX_INSTR_Z_LEN    MMIX_BYTE_BIT_LEN

#endif /* MMIXCOM_H */
```

End Listings

Frames of Reference

Michael Abrash

Several years ago, I opened a column in *Dr. Dobb's Journal* with a story about singing my daughter to sleep with Beatles songs. Beatles songs, at least the earlier ones, tend to be bouncy and pleasant, which makes them suitable good-night fodder—and there are a lot of them, a useful hedge against terminal boredom. So for many good reasons, "Can't Buy Me Love" and "Hard Day's Night" and "Help!" and the rest were evening staples for years.

No longer, though. You see, I got my wife some Beatles tapes for Christmas. We've all been listening to them in the car, and now that my daughter has heard the real thing, she can barely stand to be in the same room, much less fall asleep, when I sing those songs.

What's noteworthy is that the only variable involved in this change was my daughter's frame of reference. My singing hasn't gotten any worse over the last four years. (I'm not sure it's possible for my singing to get worse.) All that changed was my daughter's frame of reference for those songs. The rest of the universe stayed the same; the change was in her mind—lock, stock, and barrel.

Often, the key to solving a problem or working on a problem efficiently is a proper frame of reference. Your model of the problem often determines how deeply you can understand it, and how flexible and innovative you can be in solving it.

An excellent example of this, and one which I'll discuss toward the end of this column, is that of 3-D transforms—the process of converting coordinates from one coordinate space to another, for example from worldspace to viewspace. The way this is traditionally explained is functional, but not particularly intuitive, and fairly hard to visualize. Recently, I've come across another way of looking at transforms that seems far easier to grasp. The two approaches are technically equivalent, so the difference is purely a matter of how we view things—but sometimes that's the most important difference.

Before we can talk about transforming between coordinate spaces, however, we need two building blocks: dot products and cross products.

3-D Math

In my last column I promised to present a BSP-based renderer this month, to complement the BSP compiler we've developed over the last two columns. But the considerable amount of mail about 3-D math that I've received over the last two months changed my mind. In every case, the writer bemoaned his or her lack of expertise with 3-D math, asked me to recommend books about 3-D math, and questioned how they could learn more.

That's a commendable attitude, but the truth is, there's not all that much to 3-D math, at least for the sort of polygon-based, real-time 3-D done on PCs. You really need only two basic math tools beyond simple arithmetic: dot products and cross products; mostly, just the former. My friend Chris Hecker points out that this is an oversimplification; math-related operations like BSP trees, graphs, discrete math for edge stepping, and affine and perspective texture mappings also go into a production-quality game. While that's true, dot and cross products, together with matrix math and perspective projection, constitute the bulk of what most people mean by "3-D math." As we'll see, dot and cross products are key tools for a lot of useful 3-D operations.

The mail also made clear that a lot of people out there don't understand dot or cross products, at least insofar as they apply to 3-D. Since just about everything I'll do in this column relies to some extent on dot and cross products (even the line-intersection formula I discussed last time is actually a quotient of dot products), I'll devote this column to examining these basic tools and some of their 3-D applications. If this is old hat to you, my apologies; I'll return to BSP-based rendering next time.

A Little Background

Dot and cross products themselves are straightforward and require almost no context to understand, but I need to define

Michael is the author of Zen of Graphics Programming *and* Zen of Code Optimization. *He is currently pushing the envelope of real-time 3-D on* Quake *at id Software. He can be reached at mikeab@idsoftware.com.*

some terms I'll use when describing their application.

I assume you have *some* math background, so I'll quickly define a "vector" as a direction and a magnitude, represented as a coordinate pair (in 2-D) or triplet (in 3-D), relative to the origin. That's a pretty sloppy definition, but it'll do for our purposes; for the real McCoy, check out *Calculus and Analytic Geometry*, Eighth Edition, by George B. Thomas, Jr. and Ross L. Finney (Addison-Wesley, 1991, ISBN 0-201-52929-7).

So, for example, in 3-D, the vector **V**= [5 0 5] has a length, or magnitude, of $\|\mathbf{V}\|=5\sqrt{2}$, by the Pythagorean theorem, as shown in Example 1 (vertical double bars denote vector length), and a direction in the plane of the x and z axes, exactly halfway between those two axes.

I'll be working in a left-handed coordinate system, whereby if you wrap the fingers of your left hand around the z axis with your thumb pointing in the positive z direction, the fingers will curl from the positive x axis to the positive y axis. The positive x axis runs left to right across the screen, the positive y axis runs bottom to top, and the positive z axis runs into the screen.

For our purposes, projection is the process of mapping coordinates onto a line or surface. "Perspective projection" projects 3-D coordinates onto a viewplane, scaling coordinates according to their z distance from the viewpoint in order to provide proper perspective. "Objectspace" is the coordinate space in which an object is defined, independent of other objects and the world itself. "Worldspace" is the absolute frame of reference for a 3-D world; all objects' locations and orientations are with respect to worldspace, and this is the frame of reference around which the viewpoint and view direction move. "Viewspace" is worldspace as seen from the viewpoint, looking in the view direction. "Screenspace" is viewspace after perspective projection and scaling to the screen.

Finally, "transformation" is the process of converting points from one coordinate space into another; in our case, that'll mean rotating and translating (moving) points from objectspace or worldspace to viewspace.

For additional information, check out *Computer Graphics: Principles and Practice*, Second Edition, by James D. Foley and Andries van Dam (Addison-Wesley, 1990. ISBN 0-201-12110-7), or my X-Sharp columns in *DDJ* in 1992; those columns are also collected in my book *Zen of Graphics Programming* (Coriolis Group Books, 1995, ISBN 1-883577-08-X).

The Dot Product

Now for the dot product. Given two vectors $\mathbf{U}=[u_1\ u_2\ u_3]$ and $\mathbf{V}=[v_1\ v_2\ v_3]$, their dot product (denoted by the · symbol), is calculated as in Example 2(a). The result is a scalar value (a single, real-valued number), not another vector.

Now that you know how to calculate a dot product, what does that get you? Not much. The dot product isn't much use for graphics until you start thinking of it as in Example 2(b), where θ is the angle between the two vectors and the other two terms are the lengths of the vectors, as shown in Figure 1. Although it's not immediately obvious, Example 2(b) has a wide variety of applications in 3-D graphics.

Dot Products of Unit Vectors

The simplest case of the dot product is when both vectors are unit vectors; that is, when their lengths are both one, as calculated as Example 1. In this case, Example 2(b) simplifies to Example 3(a). In other words, the dot product of two unit vectors is the cosine of the angle between them.

One obvious use of this is to find angles between unit vectors, in conjunction with an inverse cosine function or lookup table. A more useful application for 3-D graphics is in lighting surfaces, where the cosine of the angle between incident light and the normal (perpendicular vector) of a surface determines the fraction of the light's full intensity at which the surface is illuminated, as in Example 3(b), where I_s is the intensity

(continued on page 50)

$$\|\mathbf{V}\|=\sqrt{v_1^2+v_2^2+v_3^2}=\sqrt{5^2+0^2+5^2}=5\sqrt{2}$$

Example 1: *The Pythagorean theorem in 3-D, where the vector* ***V***= *[5 0 5] has a length, or magnitude, of* $5\sqrt{2}$.

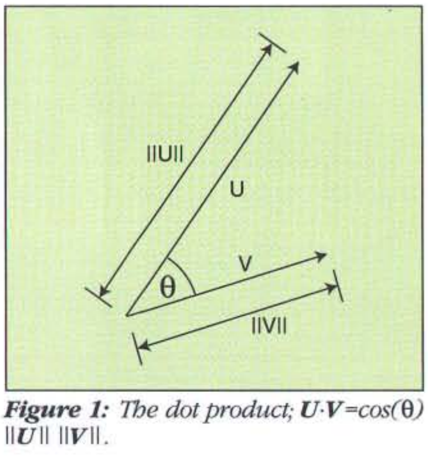


Figure 1: *The dot product;* $\mathbf{U}\cdot\mathbf{V}=\cos(\theta)\ \|\mathbf{U}\|\ \|\mathbf{V}\|$.

(a) $\mathbf{U}\cdot\mathbf{V}=u_1v_1+u_2v_2+u_3v_3$

(b) $\mathbf{U}\cdot\mathbf{V}=\cos(\theta)\ \|\mathbf{U}\|\ \|\mathbf{V}\|$

Example 2: *(a) Calculating a dot product; (b) using dot products for 3-D graphics.*

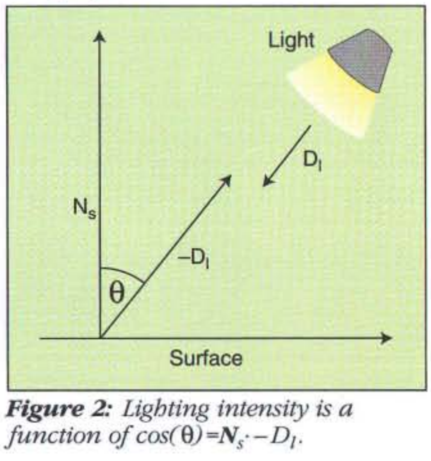


Figure 2: *Lighting intensity is a function of* $\cos(\theta)=\mathbf{N}_s\cdot-D_l$.

(a) $\mathbf{U}\cdot\mathbf{V}=\cos(\theta)$

(b) $I_s=I_l\cos(\theta)$

(c) $I_s=I_l(\mathbf{N}_s\cdot-\mathbf{D}_l)$

Example 3: *(a) Example 2(b) with unit vectors; using dot products for lighting surfaces; (c) performing the calculation with four multiplies and two additions— and no explicit cosine calculations.*

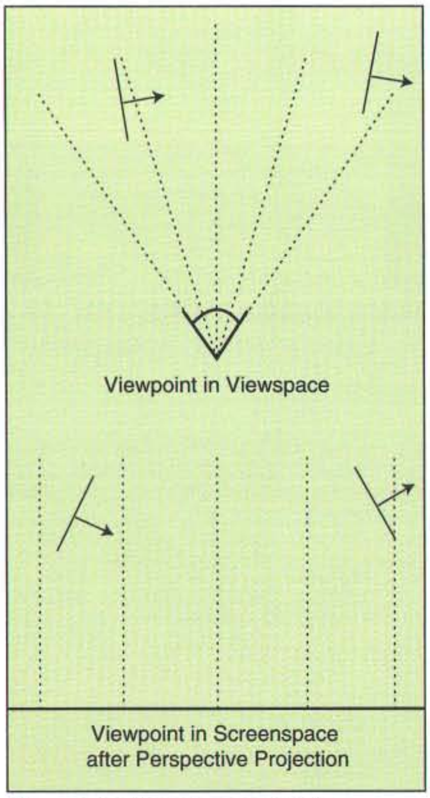


Figure 3: *Viewspace normal z direction doesn't necessarily indicate front/back visibility after perspective projection.*

NEW RELEASE!

DR. DOBB'S CD-ROM LIBRARY

You're already a Dobb's reader. Now be a Dobb's Super-User.

RUNS ON DOS & WINDOWS!

Dr. Dobb's Journal on CD-ROM RELEASE 3

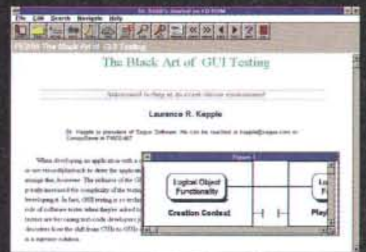
RELEASE 3 CONTAINS ALL THE ARTICLES AND SOURCE CODE FROM JANUARY 1988 THROUGH JUNE 1995—PLUS ALL SPECIAL ISSUES AND PREVIOUSLY UNPUBLISHED SOURCE CODE.

SPECIAL UPGRADE PRICING!

Buy release 3 now and get future upgrades at low prices.

EASY TO USE!

USER-FRIENDLY INTERFACE



THE DR. DOBB'S/CD EASY-TO-USE INTERFACE IS YOUR FIRST STEP TO QUICKLY FINDING THE INFORMATION YOU WANT.

FIND IT FAST!

INTUITIVE SEARCHING

FLEXIBLE SEARCH FUNCTIONS LET YOU FIND INFORMATION USING TITLE, AUTHOR, NAME, LISTINGS, AND SO ON.

SAVE TIME!

DOWNLOADABLE CODE

DON'T WASTE TIME. COPY CODE DIRECTLY INTO YOUR PROJECTS. LET THE EXPERTS DO THE WORK FOR YOU!

IN SECONDS YOU CAN ACCESS OVER SEVEN AND A HALF YEARS OF THE LEADING MAGAZINE FOR PROFESSIONAL PROGRAMMERS.

PHONE ORDERS: 800-992-0549

FAX ORDERS: 510-372-8582—USE THIS ORDER FORM

E-MAIL ORDERS: sbarnes@mfi.com

MAIL ORDERS: Dr. Dobb's/CD, P.O. Box 1525, Martinez, CA 94553-9802

INTERNATIONAL: USE MAIL, FAX, E-MAIL, OR CALL 415-655-4190

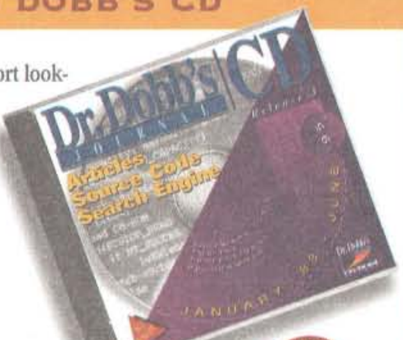
Dr. Dobb's CD-ROM LIBRARY

DR. DOBB'S CD-ROM LIBRARY

Programming expertise on demand.

Examine the next generation of languages.

DR. DOBB'S CD



Save precious time and effort looking for the answer to your current programming project! Whether it's source code or detailed documentation on a specific subject — *Dr. Dobb's/CD* has what you need.

RELEASE 3

THIS LATEST RELEASE DELIVERS ALL THE ARTICLES AND SOURCE CODE FROM *DR. DOBB'S JOURNAL* FROM JANUARY 1988 THROUGH JUNE 1995 (OVER 7 1/2 YEARS)... INCLUDING THE *DR. DOBB'S SOURCEBOOK SERIES*.

This CD-ROM puts the programming expertise and innovation of today's best programmers at your fingertips. Conduct full text searches with multiple keywords, Boolean logic, and wildcards ...in an extremely user-friendly interface. You can copy and paste, export, and print directly from the CD-ROM. It's the ultimate programming archive! Run your *Dr. Dobb's/CD* on either DOS or Windows.

ALTERNATIVE PROGRAMMING LANGUAGES CD-ROM

The editorial staff of *Dr. Dobb's Journal* went on a quest for the latest and most exciting developments in language technology — and what they found is absolutely fascinating. *The Alternative Programming Languages CD-ROM* gives you two dozen of the most useful (and exotic) languages available anywhere, packaged together in what we believe is the most extensive collection of alternative languages ever assembled.

If your work involves distributed computing or embeddable languages...or if you're simply intrigued by computer linguistics (like us)...you owe it to yourself to examine this exciting compendium. You get...

**PERL • GLISH • SMALLTALK • SATHER • TCL • MO • MODULA-3
REXX • LOUT • GHOSTSCRIPT • DYLAN • DUEL • BOB • NEUDL
PYTHON • OBERON • PARASOL • S-LANG • QUINCY • USYSTEMS
DISTRIBUTED C • THE GNU COMPILER SUITE • ...AND MORE!**

All components are robust and imaginatively designed, many break new technical ground, and each one comes with source code for compilers or interpreters. Increase your productivity...and your knowledge...with this Rosetta Stone of CD-ROMs!

Use this order form for all four CD-ROMs!

YES!

Rush me the CD-ROMs I've checked off below. I understand that I have a full money-back guarantee.

❑ Dr. Dobb's/CD-ROM Release 3	$ 79.95
❑ 386BSD Reference CD-ROM	$ 89.95
❑ Alternative Programming Languages CD-ROM	$ 49.95
❑ Essential Books on Graphics Programming CD-ROM	$ 69.95
❑ Shipping (see below)	$ ______
❑ Calif. sales tax (see below)	$ ______
Total payment:	$ ______

Shipping: $2.00 for each CD-ROM shipped in USA/Canada, $13.50 for all other countries.
(California residents must include 8.5% sales tax).

❑ Visa ❑ Mastercard ❑ American Express ❑ Check Enclosed

Credit Card #: ______________________ Expiration Date ______________

Signature ______________________________________

Name ______________________________________

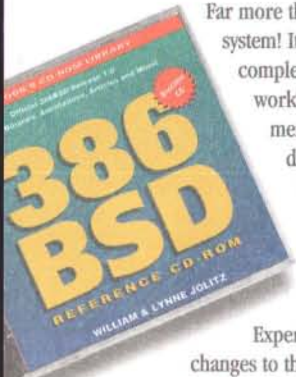
Address ______________________________________

City/State/Zip/Country ______________________________________

sB95

Glimpse the future of operating systems.

386BSD REFERENCE CD-ROM



Far more than just a high-performance operating system! It's a fully multitasking kernel containing complete interprocess communications and networking protocols (TCP/IP), advanced virtual memory subsystems, fast file systems and drivers. And it's lavishly annotated by the authors.

This CD-ROM includes hyperlinked documentation that explains key aspects of the kernel line by line. Experiment and learn with the system, make changes to the source code, recompile with the provided utilities, and see what you've created!

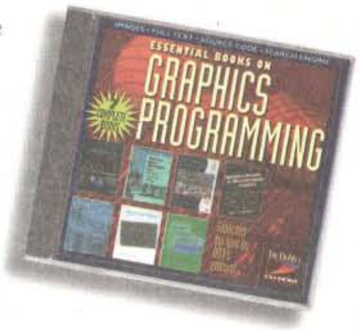
386BSD also delivers C and C++ libraries, X11R5 and X11R6, tools and utilities, and several popular application packages. There are over 575 Mbytes of source, binaries, and documentation.

Probe the details of a real operating system. Plan for the new technologies like clustering, multiprocessing, and network security. Continue your education and see the future of operating systems... today!

You can't get more graphic than this.

ESSENTIAL BOOKS ON GRAPHICS PROGRAMMING

When it comes to graphics programming, this is the one definitive source: *Dr. Dobb's Essential Books on Graphics Programming CD-ROM.*



You get seven essential books written on the subject of graphics programming...including full text, diagrams, graphics, and source code— all on one CD-ROM. From fundamental algorithms to the most complex techniques, this CD-ROM lets you find the critical information you need for graphics programming projects.

The complete books on this CD-ROM are:

- *ZEN OF GRAPHICS PROGRAMMING*, BY MICHAEL ABRASH
- *PRACTICAL IMAGE PROCESSING IN C*, BY CRAIG A. LINDLEY
- *PHOTOREALISM AND RAY TRACING IN C*, BY WATKINS, COY, AND FINLAY
- *APPLIED CONCEPTS IN MICROCOMPUTER GRAPHICS*, BY BRUCE ARTWICK
- *DIGITAL HALFTONING*, BY ROBERT ULICHNEY
- *DIGITAL IMAGE WARPING*, BY GEORGE WOLBERG
- *ALGORITHMS FOR GRAPHICS AND IMAGE PROCESSING*, BY THEO PAVLIDIS

Dr. Dobb's
CD-ROM
LIBRARY

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

Use this order form for all four CD-ROMs!

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT 31 MARTINEZ, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Dr. Dobb's CD-ROM Library
PO Box 1525
Martinez, CA 94553-9802

DR. DOBB'S CD-ROM LIBRARY

You can't get more graphic than this.

Essential Books on Graphics Programming CD-ROM

7 Complete Books on One CD-ROM!

Windows Platform

Need to get up to speed on graphics programming–fast? There's only one definitive source you can turn to–*Dr. Dobb's* Essential Books on Graphics Programming CD-ROM!

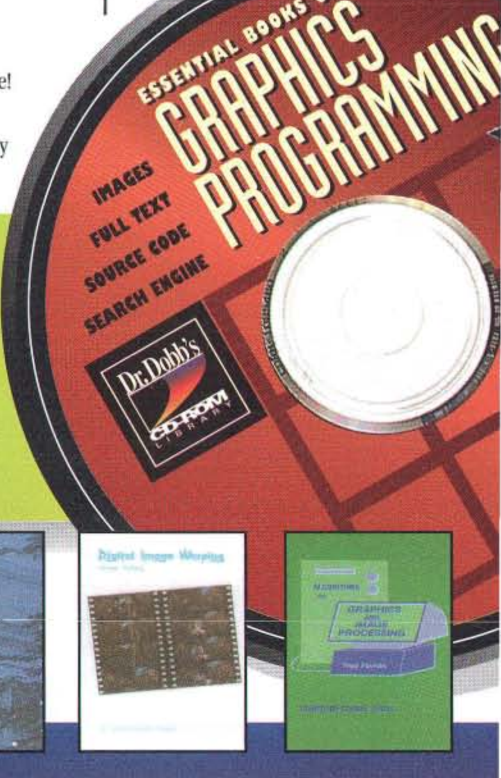
Get seven of the essential books in graphics programming, with full text, diagrams, graphics, and source code–all on one CD-ROM. From fundamental algorithms to the most complex techniques, this CD-ROM lets you find all the critical information you need for your graphics programming projects.

Texture mapping, color modeling, and morphing, all optimized for speedy 2-D and 3-D graphics programming. Practical image acquisition and processing. High-quality 3-D photorealistic graphics. Digital halftoning. Image synthesis and special effects. And much more! Plus, all the source code on the disks supplied with the books are included.

Your CD-ROM comes with must-have features:
- A powerful search engine for locating information across all or selected books
- Tables of contents and indexes hotlinked to their topics
- Copy, bookmark, and notes functions

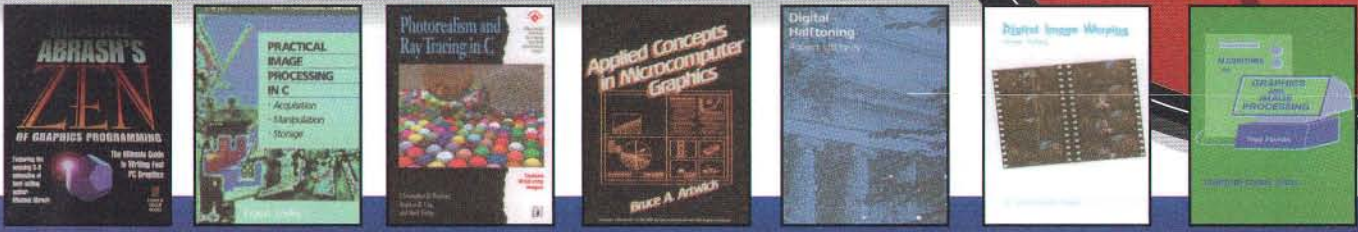
And you get a 60-day money-back guarantee!

It would cost you hundreds of dollars to buy these books separately.



THE DDJ SELECTION: WE DID THE LEGWORK FOR YOU!

- *Zen of Graphics Programming*, by Michael Abrash
- *Practical Image Processing in C*, by Craig A. Lindley
- *Photorealism and Ray Tracing in C*, by Watkins, Coy, and Finlay
- *Applied Concepts in Microcomputer Graphics*, by Bruce Artwick
- *Digital Halftoning*, by Robert Ulichney
- *Digital Image Warping*, by George Wolberg
- *Algorithms for Graphics and Image Processing*, by Theo Pavlidis


ABRASH'S ZEN OF GRAPHICS PROGRAMMING
PRACTICAL IMAGE PROCESSING IN C
Photorealism and Ray Tracing in C
Applied Concepts in Microcomputer Graphics
Bruce A. Artwick
Digital Halftoning
Digital Image Warping
GRAPHICS AND IMAGE PROCESSING

PHONE ORDERS: 800-992-0549

FAX ORDERS: 510-372-8582–USE THIS ORDER FORM

E-MAIL ORDERS: sbarnes@mfi.com

MAIL ORDERS: Dr. Dobb's/CD, P.O. Box 1525, Martinez, CA 94553-9802

INTERNATIONAL: USE MAIL, FAX, E-MAIL, OR CALL 415-655-4190

Dr. Dobb's CD-ROM LIBRARY

(continued from page 47)

of illumination of the surface, I_l is the intensity of the light, and θ is the angle between $-D_l$ (where D_l is the light direction vector) and the surface normal. If the inverse light vector and the surface normal are both unit vectors, then this calculation can be performed with four multiplies and two additions — and no explicit cosine calculations — as in Example 3(c), where N_s is the surface unit normal and D_l is the light unit direction vector; see Figure 2.

A Brief Aside on Cross Products

One question Example 3(c) begs is, Where does the surface unit normal come from? One approach is to store the end of a surface normal as an extra data point with each polygon (with the start being some point that's already in the polygon), and transform it along with the rest of the points. This has the advantage that if the normal starts out as a unit normal, it will end up that way too, if only rotations and translations (but not scaling and shears) are performed.

The problem with an explicit normal is that it will remain a normal — that is, perpendicular to the surface — only through viewspace. Rotation, translation, and scaling preserve right angles, which is why normals are still normals in viewspace, but perspective projection does not preserve angles, so vectors that were surface normals in viewspace are no longer normals in screenspace.

Why does this matter? Because, on average, half the polygons in any scene face away from the viewer, and hence shouldn't be drawn. One way to identify such polygons is to see whether they face toward or away from the viewer; that is, whether their normals have negative z values (so they're visible) or positive z values (so they should be culled). However, we're talking about screenspace normals here, because the perspective projection can shift a polygon relative to the viewpoint so that although its viewspace normal has a negative z, its screenspace normal has a positive z, and vice versa, as in Figure 3. So we need screenspace normals, but those can't readily be generated by transformation from worldspace.

The solution is to use the cross product of two of the polygon's edges to generate a normal. Example 4 is the formula for the cross product. (Note that the cross-product operation is denoted by an X.) Unlike the dot product, the result of the cross product is a vector. Not just any vector, either — the vector generated by the cross product is perpendicular to both of the original vectors. Thus, the cross product can be used to generate a normal to any surface for which you have two vectors that lie within the surface. This means that we can generate the screenspace normals we need by taking the cross product of two adjacent polygon edges, as in Figure 4. In fact, we can cull with only one-third the work needed to generate a full cross product; because we're interested only in the sign of the z component of the normal, we can skip calculating the x and y components entirely. The only caveat is to be careful that neither edge you choose is zero-length and that the edges aren't collinear, because the dot product can't produce a normal in those cases.

Perhaps the most often asked question about cross products is, Which way do normals generated by cross products go? In a left-handed coordinate system, curl the fingers of your left hand so the fingers curl through an angle of less than 180 degrees from the first vector in the cross product to the second vector. Your thumb now points in the direction of the normal.

If you take the cross product of two orthogonal (right-angle) unit vectors, the result will be a unit vector that's orthogonal to both of them. This means that if you're generating a new coordinate space — such as a new viewing frame of reference — you only need to come up with unit vectors for two of the axes for the new coordinate space. You can then use their cross product to generate the unit vector for the third axis. If you need unit normals and the two vectors being crossed aren't orthogonal unit vectors, you'll have to normalize the resulting vec-

The dot product is the cosine of the angle between two vectors

$$\mathbf{U}\times\mathbf{V}=[u_2v_3-u_3v_2 \;\; u_3v_1-u_1v_3 \;\; u_1v_2-u_2v_1]$$

Example 4: *Formula for the cross product.*

(a) $\mathbf{U}\cdot\mathbf{V}=\cos(\theta)\,||\mathbf{U}||$

(b) $\text{distance}=|(\mathbf{P}-\mathbf{O}_p)\cdot\mathbf{N}_p|$

Example 5: *(a) Using the dot product for projection; (b) using the dot product to determine the distance to a plane.*

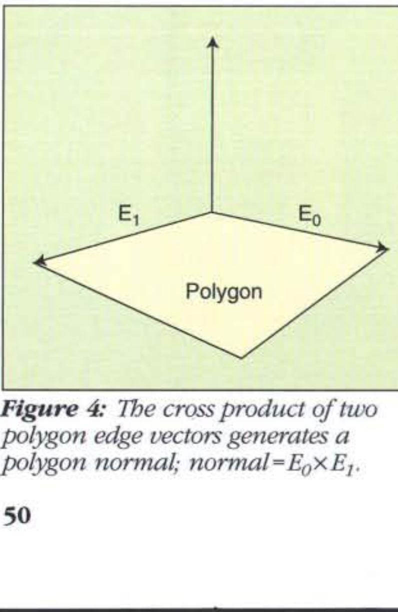


Figure 4: *The cross product of two polygon edge vectors generates a polygon normal; normal=$E_0\times E_1$.*

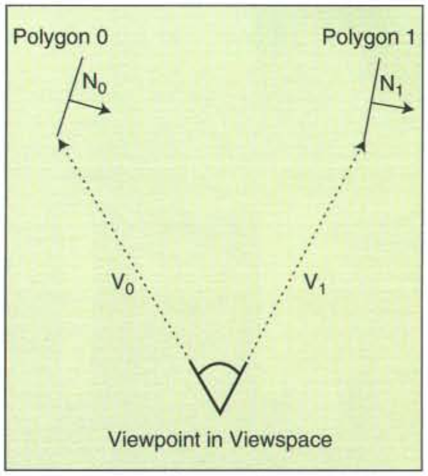


Figure 5: *Backface culling with the dot product. $V_0\cdot N_0<0$, so polygon 0 faces forward and is visible; $V_1\cdot N_1>0$, so polygon 1 faces backward and is invisible.*

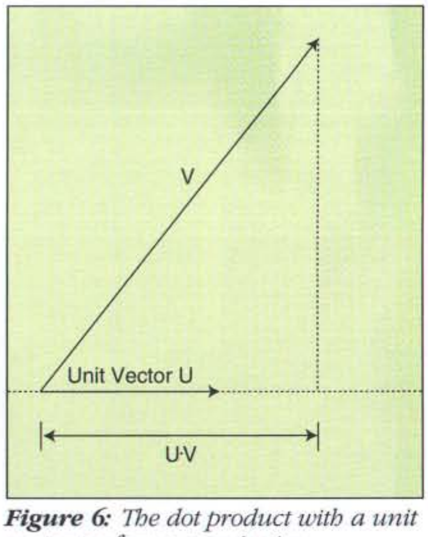


Figure 6: *The dot product with a unit vector performs a projection.*

tor; that is, divide each of the vector's components by the length of the vector, to make it a unit long.

Using the Sign of the Dot Product

The dot product is the cosine of the angle between two vectors, scaled by the magnitudes of the vectors. Magnitudes are always positive, so the sign of the cosine determines the sign of the result. The dot product is positive if the angle between the vectors is less than 90 degrees, negative if it's greater than 90 degrees, and 0 if the angle is exactly 90 degrees. This means that just the sign of the dot product suffices for tests involving comparisons of angles to 90 degrees, and there are more of those than you'd think.

Consider, for example, the process of backface culling, discussed earlier in the context of using screenspace normals to determine polygon orientation relative to the viewer. The problem with that approach is that it requires each polygon to be transformed into viewspace, then perspective projected into screenspace, before the test can be performed, and that involves a lot of time-consuming calculation. Instead, we can perform culling way back in worldspace (or even earlier, in objectspace, if we transform the viewpoint into that frame of reference), given only a vertex and a normal for each polygon and a location for the viewer.

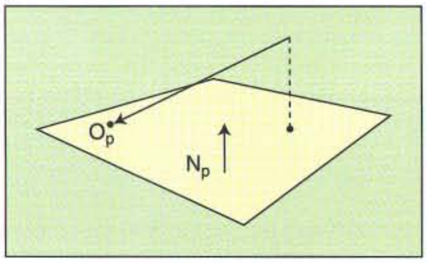


***Figure 7:** Using the dot product to get the distance from a point to a plane; distance =* $|(\boldsymbol{P}-\boldsymbol{O}_p)\cdot\boldsymbol{N}_p|$.

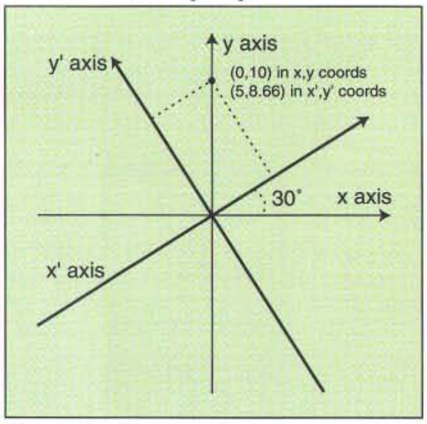


***Figure 8:** Rotating to a new coordinate space by projection onto the new axes.*

The simplest case of the dot product is when both vectors are unit vectors

Here's the trick: Calculate the vector from the viewpoint to any vertex in the polygon, and take its dot product with the polygon's normal, as in Figure 5. If the polygon is facing the viewpoint, the result is negative, because the angle between the two vectors is greater than 90 degrees. If the polygon is facing away, the result is positive, and if the polygon is edge-on, the result is 0. That's all there is to it—and this sort of backface culling happens before any transformation or projection at all is performed, saving a great deal of work for the half of all polygons, on average, that are culled.

Backface culling with the dot product is just a special case of determining which side of a plane any point (in this case, the viewpoint) is on. The same trick can be applied to determine whether a point is in front of or behind a plane, where a plane is described by any point that's on the plane (which I'll call the "plane origin"), plus a plane normal. One such application is in clipping a line (such as a polygon edge) to a plane. Just do a dot product between the plane normal and the vector from one line endpoint to the plane origin, and repeat for the other line endpoint. If the signs of the dot products are the same, no clipping is needed; if they differ, it is. And yes, the dot product is also the way to do the actual clipping; but before we can talk about that, we need to understand the use of the dot product for projection.

Using the Dot Product for Projection

Consider Example 2(b) again, but this time making one of the vectors, say **V**, a unit vector. Now the equation reduces to Example 5(a). In other words, the result is the cosine of the angle between the two vectors, scaled by the magnitude of the nonunit vector. Now, consider that cosine is really just the length

```
// Given two line endpoints, a point on a plane, and a unit normal
// for the plane, returns the point of intersection of the line
// and the plane in intersectpoint.

#define DOT_PRODUCT(x,y)   (x[0]*y[0]+x[1]*y[1]+x[2]*y[2])

void LineIntersectPlane (float *linestart, float *lineend,
   float *planeorigin, float *planenormal, float *intersectpoint)
{
   float vec1[3], projectedlinelength, startdistfromplane, scale;

   vec1[0] = linestart[0] - planeorigin[0];
   vec1[1] = linestart[1] - planeorigin[1];
   vec1[2] = linestart[2] - planeorigin[2];

   startdistfromplane = DOT_PRODUCT(vec1, planenormal);

   if (startdistfromplane == 0)
   {
        // point is in plane
        intersectpoint[0] = linestart[0];
        intersectpoint[1] = linestart[1];
        intersectpoint[2] = linestart[1];
        return;
   }

   vec1[0] = linestart[0] - lineend[0];
   vec1[1] = linestart[1] - lineend[1];
   vec1[2] = linestart[2] - lineend[2];

   projectedlinelength = DOT_PRODUCT(vec1, planenormal);

   scale = startdistfromplane / projectedlinelength;

   intersectpoint[0] = linestart[0] - vec1[0] * scale;
   intersectpoint[1] = linestart[1] - vec1[1] * scale;
   intersectpoint[2] = linestart[1] - vec1[2] * scale;
}
```

***Example 6:** The intersection point on a line segment.*

of the adjacent leg of a right triangle, think of the nonunit vector as the hypotenuse of a right triangle, and remember that all sides of similar triangles scale equally. What it all works out to is that the value of the dot product of any vector with a unit vector is the length of the first vector projected onto the unit vector, as in Figure 6.

This unlocks all sorts of neat stuff. Want to know the distance from a point to a plane? Just dot the vector from the point P to the plane origin O_p with the plane unit normal N_p, to project the vector onto the normal, then take the absolute value, as shown in Example 5(b) and Figure 7.

Want to clip a line to a plane? Calculate the distance from one endpoint to the plane, as just described, and dot the whole line segment with the plane normal, to get the full length of the line along the plane normal. The ratio of the two dot products is then how far along the line from the endpoint the intersection point is; just move along the line segment by that distance from the endpoint, and you're at the intersection point, as shown in Example 6.

Rotation by Projection

You can use the dot product's projection capability to look at rotation in an interesting way. Typically, rotations are represented by matrices. This is certainly a workable representation that encapsulates all aspects of transformation in a single object, and it is ideal for concatenations of rotations and translations. One problem with matrices, though, is that many people, myself included, have a hard time looking at a matrix of sines and cosines and visualizing what's actually going on. So when two 3-D experts, John Carmack and Billy Zelsnack, mentioned that they think of rotation differently, in a way that seemed more intuitive to me, I thought it was worth passing on.

Their approach is this: Think of rotation as projecting coordinates onto new axes. That is, given that you have points in, say, worldspace, define the new coordinate space (viewspace, for example) to which you want to rotate by a set of three orthogonal unit vectors defining the new axes, and then project each point onto each of the three axes to get the coordinates in the new coordinate space, as shown for the 2-D case in Figure 8. In 3-D, this involves three dot products per point, one to project the point onto each axis. Translation can be done separately from rotation by simple addition.

Rotation by projection is exactly the same as rotation via matrix multiplication; in fact, the rows of a rotation matrix are the orthogonal unit vectors pointing along the new axes. Rotation by projection buys us no technical advantages, so that's not what's important here; the key is that the concept of rotation by projection, together with a separate translation step, gives us a new way to look at transformation that I, for one, find easier to visualize and experiment with. A new frame of reference for how we think about 3-D frames of reference, if you will.

Three things I've learned over the years are that:

- It never hurts to learn a new way of looking at things.
- It helps to have a clearer, more intuitive model in your head of whatever it is you're working on.
- New tools, or new ways to use old tools, are Good Things.

My experience has been that rotation by projection, and dot-product tricks in general, offer those sorts of benefits for 3-D.

Next time, we'll do BSP-based rendering, and if there's room, maybe I can sneak in a sample app that shows some smart dot tricks in action.

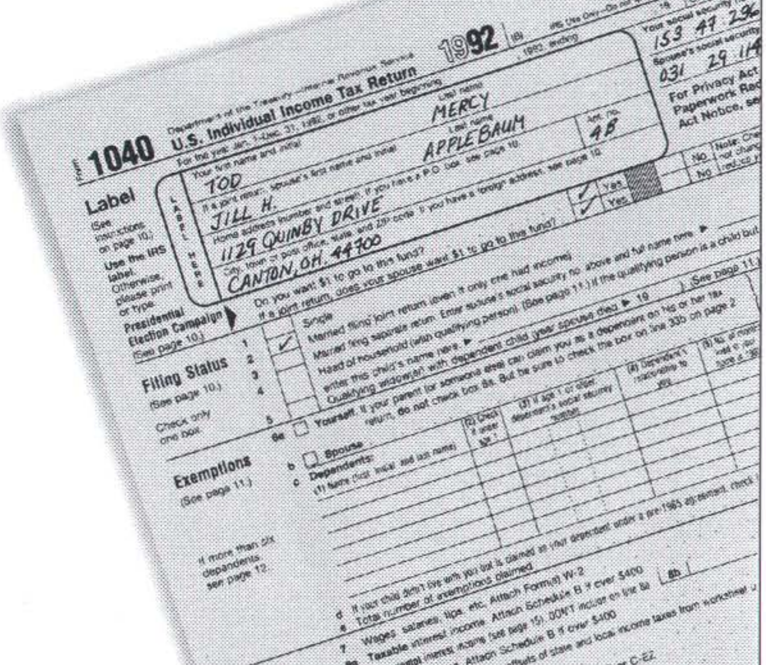
DDJ

Please don't let this be the only blood you give this year.

Photo: Sandra Eisner

American Red Cross

Give blood again. It will be felt for a lifetime.


1040 U.S. Individual Income Tax Return 1992

Label

TOD

JILL H.

MERCY

APPLEBAUM

4B

1129 QUINBY DRIVE

CANTON, OH 44700

153 47 236

031 29 114

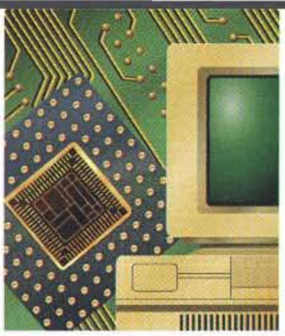
Presidential Election Campaign

Filing Status

Exemptions

Income

Ad Council



Rocket Science Made Simple

Hal W. Hardenbergh

I'm going to explain some important stuff about computer architecture, stuff that you really need to know. I'll cover the Pentium, PowerPC 601, and the P6. We have to discuss a few basics before we come to the important stuff.

The term "computer architecture" is widely misunderstood. It has little to do with the design of a computer system or microprocessor chip. The computer architect is best known as the person who gets to use a clean piece of paper to define which instructions the computer will be able to execute. But the most important job the architect does is decide on the length(s), in bits, of the computer instructions and assign the bit fields within that length to perform the necessary computer operations.

If there's a large proportion of unused combinations, the architect has done a lousy job. But a few should be set aside. When Intel designed the 8086, some then-undefined combinations later became the basis for adding a (very) few more registers in the 386 generation.

The 8086 was designed back when 64K was a *huge* memory space and Pascal seemed to be taking over the personal-computer marketplace. So the 8086 was given exactly enough registers to run compiled Pascal.

Because memory was then an extremely limited resource, the 8086's basic instruction-field length was made eight bits, and some of Pascal's most common instructions (LOOP, for example) were fitted into those eight bits; eight bits does not provide for specifying a lot of registers.

When the 68000 was designed, larger memories were common, so the architect selected a 16-bit basic instruction field. Two 4-bit register fields were assigned. Eight bits, half the 16-bit instruction field, went to defining the source and destination registers.

Hal is a hardware engineer who sometimes programs. He is the former editor of DTACK Grounded *and can be contacted through the* DDJ *offices.*

But the ability to place many transistors on a single die was exploding, and soon 32 each, 32-bit registers started showing up, for instance on David Patterson's Berkeley RISC I design. Five bits are required to select one of 32 registers. If two-address (SRC, DEST) operands were to be used, then ten bits of the instruction bit field were needed to specify the registers. That leaves only six bits of a 16-bit instruction field, not enough to be useful.

So computers with 32 registers moved up to a 32-bit instruction field. All the computer architects made the decision to use three-address operands (SRC1, SRC2, DEST) and so assigned 15 bits just for register selection—again, about half the instruction field.

The microprocessor went from a register-starved, 8-bit instruction field in 1977 to a register-rich, 32-bit instruction field in 1982. These architectural decisions were dictated by the then state of the chip-fabrication art. Let me repeat—these were architectural decisions.

And architectural innovations stopped right there in 1982, because a personal computer does not (yet) need a 64-bit instruction field. Yep. Architecture for personal computers essentially froze in 1982.

How do you upgrade a computer to a new architecture? In other words, how do you get your hands on more registers while continuing to run your old software? The answer is, you don't. The only way to get more registers is to abandon your software—*all* your software—and move to a new computer. I understand the MIPS-based ACE computer systems (which run both UNIX and Windows NT) are particularly good examples of desktop computers with register-rich environments.

Oh? You don't have an ACE system on your desktop? You still use, and program for, a register-starved computer architecture? Gee. *It appears that computer architecture, while fundamental, is not important.*

The personal-computer marketplace doesn't care about architectural hardware issues. The marketplace responds to fast and cheap. "Fast" means internal caches, floating-point accelerators, superscalar techniques, and the like—none of which has anything to do with architecture. (The presence or absence of an internal cache is independent of the instruction field.)

"Cheap" means economy of scale. More than 50 million personal computers will be sold this year, and to a first-order approximation, 100 percent of them will be based on the x86 architecture. If you want a cheap computer, buy one based on the x86.

But the marketplace still wants to run the software it acquired ten years ago. *Software* compatibility is, in fact, an architectural issue, *and it matters in the marketplace.*

The people who designed the Pentium and the P6 and who are currently designing the P7 are *not* computer architects. But they're pretty good engineers, based on the results I've seen. I call them "chip designers."

Back when the world was young and children were respectful of their elders, the chip designer's job was simple: The design had to execute any instruction as quickly as possible. Then it had to execute the next instruction as quickly as possible. That's how the 8086, 286, 386, and 486 work.

But with the advent of the Pentium, those days are gone. The Pentium—sometimes—executes more than one instruction in the same clock cycle. That "sometimes" is pretty important to those of you who need to write code that runs fast, and has afforded my colleague Michael Abrash the opportunity to publish several articles on optimizing code for the Pentium.

The Pentium is the first x86 generation that uses a "superscalar" implementation. Let's compare it to the PowerPC 601, which was primarily designed by IBM, with a little bus-interface assistance from Motorola. To a first-order approximation, the 60x architecture has 0 percent of the personal-computer market.

The 601 is based on the latest computer architecture: the 32-bit model with 32 registers. Like the Pentium, its implementation uses superscalar techniques, but not those used by the Pentium. The 601 can issue up to three instructions each clock cycle, one each of integer, floating point (fp), and branch.

You are the software experts, not me, so let's pretend *you* just explained to me that most application programs in the personal-computer market execute instructions in the ratio 85 percent integer, 0 percent fp, and 15 percent branch. This means the 601's ability to simultaneously execute fp instructions with integer and branch instructions is useless. The only improvement the superscalar 601 offers is the ability to simultaneously issue integer and branch instructions. And since there are roughly six times as many integer as branch instructions, this isn't terribly useful. In fact, the 601's superscalar ability means that, at best, it can execute 100 instructions in 85 clocks (assuming one clock per instruction). All that superscalar design effort provides, at best, a 17.6-percent performance improvement.

The Pentium's designers were much more crude. If either an fp or branch instruction is issued on a given clock cycle, then no other instruction can be issued at that time. In practice, this means that during the 15 percent of the time that branch instructions are being issued, the Pentium ain't superscalar. But in the 85 percent of the time that integer instructions are being issued, the Pentium can—sometimes—issue two integer instructions on the same clock cycle. This means the Pentium can, at best, execute a 100-instruction mix (assuming one clock per instruction cycle) in 85/2+15=57.5 clocks—a 73.9 percent performance improvement.

Okay, instructions sometimes need more than a single clock to execute, and the Pentium cannot always issue two integer instructions in the same clock period, thus Abrash's fine articles on optimization. But Intel's chip designers focused on improving performance during the 85 percent of the time that integer instructions are being issued, while IBM's designers concentrated their efforts on the 15 percent of the time that branch instructions were being issued.

Which design team best earned its paycheck?

I sent a copy of the penultimate draft of this article to some folks who used to design microprocessor chips for a living. One of them, John Wharton, called me back and said "Hal, the Pentium doesn't work like that!" (The last four digits of John's home phone number are 8051, which is one of Intel's most popular 8-bit micros.)

So I was wrong. A Pentium can issue a branch instruction after an integer instruction in the same clock (but not an integer instruction after a branch instruction). And under rare circumstances the Pentium can issue two FP instructions in the same clock—if one of them is an FXCH instruction.

In the pairing rules, a "complex" instruction is a microprogrammed instruction, such as one of the string instructions (MOVS or SCAS, for example). When one of the integer pipes goes into microprogrammed mode, both pipes do. That's why only one "complex" instruction can be active at a time.

John also explained floating-point processing:

> A cute trick the Pentium designers came up with was getting the result of a 64-bit FP operation back to the internal cache quickly. FP operations use the integer pipes, each of which is 32 bits wide. So the Pentium uses both pipes to move 64 bits in parallel. It saves one clock and at Pentium speeds, one clock is important.

(The most interesting thing John told me was about the infighting—I call it civil war—over Intel's upcoming P7. But that's another story.)

The Pentium design team set up two on-chip production lines, like Ford using one line for Escorts and another for Taurii. With a budget of 5.5 million transistors, the P6 design team was able to use more advanced techniques. Continuing with the automotive analogy, the P6 makes intensive efforts to build a car in the shortest time.

In the P6, we find a large crowd gathered at the input ends of several parallel production lines (pipes), and another large crowd at the output ends.

The input crowd looks for tasks ready to proceed and issues them to one of the production lines. It also looks for tasks that *might* be ready to proceed and speculatively issues them, too. A list of 30 tasks to select from is kept.

The crowd at the output accepts and temporarily stores all the results the several production lines deliver. Not everything that comes off the production lines proves to be useful. Some "product" is ultimately discarded. ("We can't use that blue trunk assembly on this red car, Fred. Throw it away!")

A scoreboard keeps track of everything that's going on. The P6 has a lot more registers than the programmer's model asserts, and renames them for efficiency. How did Intel's designers get so smart? They probably read Chaitin et al.'s tutorial, "Register Allocation via Coloring," which is part of the June 1982 SIGPLAN Proceedings on compiler construction. Yes, *tutorial*. In 1982. You didn't think this stuff was new, did you?

> [Abstract: *Register allocation may be viewed as a graph coloring problem... Preliminary results... suggest that global register allocation approaching that of hand-coded assembly language may be attainable.*]

Now you should have a grasp of what Intel means when it says the P6 uses scoreboarding techniques and issues instructions speculatively. Specifically, the P6 guesses which branch paths will be taken and speculatively executes the instructions following those branches (assuming no data dependencies). If those branches are taken, then the instruction results are already available. Otherwise, the results are discarded. The P6 speculatively executes instructions passed up to five (!) branches, assuming they're available in the 30-instruction queue at the front end.

The P6 is Intel's first x86 that does not always directly execute x86 instructions. If you've read Abrash's articles on Pentium optimization, you know the performance benefits of breaking some complex instructions down into two simpler, yet equivalent, x86 instructions. Well, the P6 takes this a step further. The P6's instruction decoder will often break a complex x86 instruction into simpler instructions, that may not be x86 instructions at all.

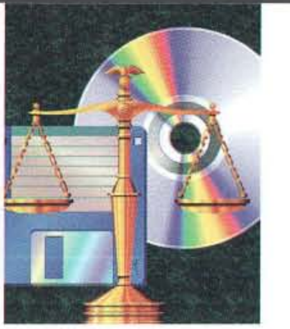
Since P6 continually looks at the next 30 instructions and begins execution of each as soon as possible, and automatically breaks up complex instructions when beneficial, you won't have to *optimize P6 code.*

The P6 self-optimizes all that shrink-wrapped code, no matter what generation of optimizing compiler was used. Poor Michael Abrash! He'll have nothing to write about, and the bank will foreclose his mortgage.

The philosophical design differences underlying the 486, Pentium, and P6 generations have nothing whatever to do with computer architecture and everything to do with chip design. The best chips are designed by persons familiar with happenings in the mainframe and minicomputer arenas a dozen or more years back.

Intel's Andrew Grove once publicly asserted that there wasn't any use for a million-transistor-plus chip except for memory. If he'd known his x86 chip designers would soon be crafting microprocessors that performed useless instructions and wouldn't even directly execute x86 code, do you suppose he'd have fired them?

DDJ



Patents: Best Protection for Software Today?

Marc E. Brown

Patents and software used to be words that were not spoken of together. Today, that's changing— and fast. Recent court decisions have expanded the availability of patents for software. Furthermore, the U.S. Patent Office has just gotten in line by issuing new proposed guidelines for the examination of "computer-implemented" inventions. The procedures for enforcing software patents are also being trimmed down to make them less expensive and faster.

At the same time, other forms of legal protection for software are becoming less attractive. Copyrights provide only limited protection. This is known only too well to Lotus, which recently was told by an appellate court that the copyrights on its famous 1-2-3 spreadsheet program do not bar Borland from copying the entire 1-2-3 menu tree. Trade-secret protection is also often lost inadvertently, particularly with publicly distributed software.

Unless you are planning to develop your software in a cave during the next decade, therefore, you need to be able to determine when software can be patented, how to patent it, and how to deal with patent-infringement allegations. Let's begin by taking take a look at three tests for patentability.

Statutory Subject Matter

I usually like to leave out the legal jargon in my column. But if you want to be knowledgeable in this area, remember the phrase "Statutory subject matter." This concept addresses whether the "subject matter" of the invention is listed in the "statute" (35 U.S.C. §101) that defines the types of inventions entitled to a patent. It is this requirement that has developed into an impediment to software patents.

This statute states that a patent may be obtained on "any new and useful process, machine, manufacture, or composition of matter, or any new and useful improvement thereof...." Software clearly falls within at least one of these broad areas. Nevertheless, the U.S. Supreme Court refused a patent on a software-driven process for converting a BCD number into pure binary. In its 1972 decision of *Gottschalk v. Benson*, the Court said that a patent could not be obtained on "laws of nature, physical phenomena and abstract ideas."

Again in 1978, the Supreme Court refused to issue a software-related patent, this time on a method of updating numerical alarm limits using a computer. In *Parker v. Flook*, the Court expressed concern that such a patent consisted merely of a new formula and the computer that implemented it.

But in 1981, the Supreme Court changed tacks. In *Diamond v. Diehr*, the Court approved a patent on a process for curing rubber that implemented a well-known mathematical equation (the Arrhenius equation) in a computer to calculate optimum cure time. Although the Court reiterated that laws of nature, natural phenomena, and abstract ideas are not patentable, it said that a patent could be granted on a practical application of a concept, even if it included a programmed, digital computer.

At about the same time, Congress created a new court to handle all appeals in patent cases. It is called the "United States Court of Appeals for the Federal Circuit." In a series of recent decisions, the Federal Circuit has made clear that patents may be granted on a broad variety of inventions containing software.

Perhaps most important is its 1994 decision in *In Re Alappat*. *Alappat* involved a software program that implemented a series of algorithms to clarify the picture that is displayed on an oscilloscope. In support of its conclusion that such an invention was "statutory," the Federal Circuit stated that the invention was "not a disembodied mathematical concept..., but rather a specific machine to produce a concrete and tangible result." The court noted that "a general-purpose computer in effect becomes a special purpose computer once it is programmed to perform particular functions pursuant to instructions from program software...."

About two weeks later in *In Re Warmerdam*, the Federal Circuit ruled that a patent could be issued in connection with an algorithm that created a data structure that controlled moving objects, such as robots, so that they would not collide with other objects.

On July 31, 1995, the U.S. Patent and Trademark Office followed suit by issuing proposed guidelines for "computer-implemented" inventions. "These guidelines respond to recent changes in the law that govern the patentability of computer-implemented inventions," the Office said. The proposed guidelines essentially provided that all software-related inventions constitute patentable subject matter, except when the invention falls within one of the following categories:

- A compilation or arrangement of data, independent of any physical elements.
- A known, machine-readable storage medium encoded with data representing creative or artistic expression (for example, a work of music, art, or literature).
- A "data structure" independent of any physical element (that is, not as implemented on a physical component of a computer such as a computer-readable memory to render that component capable of causing a computer to operate in a particular manner).

Marc is a patent attorney and shareholder of the intellectual-property law firm of Poms, Smith, Lande & Rose in Los Angeles, CA. Marc specializes in computer law and can be contacted on CompuServe at 73414,1226.

- A process that does nothing more than manipulate abstract ideas or concepts (for example, a process consisting solely of the steps one would follow in solving a mathematical problem).

Significantly, none of the excluded categories appear to embrace a general-purpose computer running a software program. Does this mean that all software can be patented by simply claiming a general-purpose computer running the new software program? The proposed guidelines appear to address this question in the following way:

> [I]n rare situations, a claim classified as a statutory machine or article of manufacture may define nonstatutory subject matter. Nonstatutory subject matter (i.e., abstract ideas, laws of nature, and natural phenomena) does not become statutory merely through a different form of claim presentation.
>
> Such a claim will (a) define the "invention" not through characteristics of the machine or article of manufacture claimed but exclusively in terms of a nonstatutory process that is to be performed on or using that machine or article of manufacture, and (b) encompass any product in the stated class (e.g., computer, computer-readable memory) configured in any manner to perform that process.

To avoid this exclusion, it seems as though the invention must be defined "through characteristics of the machine or article of manufacture claimed." But what does this mean? Is this different from the second requirement that the invention not "encompass any product in the stated class?" I really don't know! But I do know that the guidelines are intended to conform the practices of the Patent Office with the more liberal views of the courts. If the guidelines are interpreted to preclude patents on all general-purpose computers running new software, that goal seemingly will not be reached.

These guidelines may already have been clarified by the time you read this column. They were issued in June of this year for "public comment." Final guidelines were promised for July 31, 1995. Check my next column for an update.

Novelty

The second requirement for a patent is that the invention be "novel." This is usually a very easy test to pass. It simply means that the invention is *in some way new*. Any distinction whatsoever from what was done before is sufficient.

An invention's "novelty" is usually lost in the United States if an application for patent is not filed within one year after certain activity has begun. (Many foreign countries have no such grace period.)

"Nonobviousness" does not mean that the software has to be great or that it has to achieve a remarkable result

Three major types of activity usually start the one-year clock.

- When a product embodying the invention is offered for sale by anyone, even someone other than the inventor. The offer need not result in an actual sale.
- When the invention is described in a "printed publication." There is no requirement that the publication be widely distributed. Indeed, papers distributed at conferences are usually sufficient.
- When the invention is used for its intended purpose in a nonexperimental environment. If you invent a superior database system and allow your spouse to use it to manage the groceries, you had better start the one-year clock. Even your own use of the invention can start the clock running. The clock will usually not begin while the primary purpose of the use is to determine whether the invention works.

Nonobviousness

The third requirement for obtaining a patent is that the invention not be "obvious" in view of what has been done before. This is the *only* qualitative test that is applied.

"Nonobviousness" does not mean that the software has to be great or achieve a remarkable result. It merely means that the underlying concepts of the software are not obvious in view of what was known before.

Most software is simply combinations of previously known routines. In determining "obviousness," therefore, the real question is whether it would have been obvious to have combined these routines to make the software.

The determination of "obviousness" is necessarily subjective. But several objective factors will be considered:

- The art taught away from the approach that the software took.
- Widespread efforts to obtain the benefits of the software were previously unsuccessful.
- The software has received widespread recognition.
- The software achieves new and unexpected results.
- The software has been a commercial success.
- Nothing in the prior art suggests combining the routines contained in the software.

The Patent-Application Process

The first step in patenting your software is to document it by preparing block diagrams, flowcharts, specifications, data-structure maps, screen layouts, and the like. While source code is obviously the most precise formulation of the software, it is usually not perfected until well after the software is conceived. Also, it does not usually communicate the broad concepts implemented by the software.

It's wise to corroborate the date on which the documentation was prepared. At the very least, it should be signed and dated by the developer. It is also good practice to have people not involved with the development read, date, and sign the documentation. Above each signature, the document should state that the witness has read and understood the information, as well as the number of pages it contains. It is also a good idea to keep a bound invention notebook and to date each new entry.

A patentability search is usually performed next. Although not required by law, it is very useful, as it may reveal that the software is not sufficiently different from previous work to justify the expense of a patent. Even when you are sure about the distinctiveness of your new software, knowledge of the closest prior art will help to frame the patent application in the broadest possible way.

The next step is to prepare the application. A patent application usually contains drawings, a written description of the invention, and a set of "claims" (English descriptions of the invention's elements). You are not required to build a working model of your invention before filing the application.

Unless you have considerable experience with patents, it is unlikely that you will be able to prepare the application yourself. Normally, applications are prepared by a patent attorney or patent agent, both of whom must first pass a proficiency test. But, you can do certain things to assist in its preparation:

- You are legally bound to provide the attorney or agent with copies or a description of the closest prior art of which you are aware.
- Identify the specific differences— steps, components, results— between your software and the prior art.
- Disclose the "best mode" you are aware of for implementing your invention. If you don't disclose certain features with the thought of keeping them secret and this is later discovered, your patent will be declared invalid. For marketed software this is usually easy to prove. To easily satisfy the "best mode" requirement, provide the Patent Office with a complete copy of your source code. If this makes you feel uneasy, a patent may not be for you!
- Include sufficient information to enable a person of ordinary skill in the art to which your invention pertains to make and use the invention without undue experimentation. Submitting the source code will often fulfill this requirement, too. The more detail, the better. The only downside is the cost of documenting such detail.

After your application is filed, it will be assigned to a Patent Office "examiner"— a person knowledgeable in the field of your invention and in the principles of patent law.

Approximately six months to a year after the application is filed, you will receive an "office action," a written response to your application from the Patent Office examiner. The office action will either allow your application or explain why it is being rejected.

Rejected applications may be amended to try and overcome the grounds of rejection. Alternatively, you can argue that the rejection is unjustified.

A second rejection is usually final. You will usually have to pay an additional fee to submit further amendments or arguments. You can also appeal the examiner's final rejection to the Board of Patent Appeals and, as thereafter necessary, to the federal courts.

Enforcement

The first step in enforcing a patent is to determine whether the software of the accused party is infringing.

Those unfamiliar with patent law usually don't make this determination correctly. Inventors often feel that there is an infringement when the competing software incorporates features described in the patent. Accused infringers, on the other hand, often conclude that there is no infringement because their software does not contain *every* feature described in the patent.

Neither approach is correct. The drawings and detailed descriptions in a patent are usually merely examples of the invention, not the invention itself. Not utilizing every feature in an example does not necessarily avoid infringement. Conversely, using a few of the features does not necessarily imply infringement.

The true test of infringement is whether the software in question contains every feature documented in any single claim at the end of the patent. This is the rule: A patent is infringed when the software in question contains every element recited in any single patent claim.

When determining the scope of each element, the words describing it should be given their ordinary meaning in the art, except when a contrary meaning is expressed in the patent. The words should also be given their broadest reasonable meaning, not restricted to the specific examples described in the patent.

In three circumstances, an infringement will be found, even if the infringer's software does not contain all of the claim's elements.

- The software contains the equivalent of each missing element, usually a corresponding element that performs substantially the same function in substantially the same way, to achieve substantially the same result. The precise reach of this "doctrine of equivalents" is expected to be the subject of a decision by the Federal Circuit in *In Re Hilton*.
- The missing elements are found in the computer system in which the infringer's software is installed. If the software has no substantial use other than in a system that infringes the patent claim, the person making or selling the software will usually be liable as a "contributory infringer."
- The accused infringer did not actually commit the infringement, but encouraged the person who did by distributing promotional material or product manuals that promote the software as useful in a configuration that infringes the patent claim. This is known as "inducing infringement." An officer or employee of a company who actively participates in infringing activity of that company can also be held personally liable under this theory.

Charges of infringement should be made carefully, as they give the alleged infringer the right to sue the person charging infringement for a "declaratory judgment" that the patent is not infringed or is not enforceable. Unless defended at typically great expense, the patent could be lost.

It is particularly risky to charge customers of a manufacturer with infringement. If the claim turns out to be unmeritorious, the person charging infringement can be exposed to counterclaims for libel, slander, disparagement, interference with contract, and violation of the antitrust laws.

Responding to a charge of patent infringement requires even greater care. All too often, the accused infringer denies the infringement allegation without having the allegation analyzed by a competent patent attorney. Be warned: A company that continues infringing activity without having first received a favorable legal opinion will often be assessed treble damages and attorney's fees if it loses the case.

Patent litigation has traditionally been very expensive, but this may also be changing. The Federal Circuit just held in *Markman v. Westview Investments* that disputes over the scope of a patent should be determined by a judge, not a jury. Thus, many cases will now be resolved far short of an expensive jury trial.

The right to a jury trial in patent cases is also now being questioned. In *American Airlines v. Lockwood*, the Supreme Court has agreed to decide whether the U.S. Constitution gives an alleged infringer a right to a jury trial. Abolishing jury trials in patent cases entirely would result in additional savings.

In many cases, the alleged infringer is aware of prior art that is closer to the invention than that which the Patent Office knew about when it was issued. He may then challenge the validity of the patent, arguing that the invention was "obvious." Although this can be done in court, it also can often be done in a separate Petition for Reexamination in the Patent Office. Seeking reexamination of the patent in the Patent Office is far less expensive than court litigation. Unfortunately, the alleged infringer is usually not permitted to participate during reexamination. Therefore, alleged infringers who have the financial resources often opt to have their invalidity allegation determined by a court.

Conclusion

Considerable dispute continues over the type of software entitled to a patent. While some are arguing, others are applying for and receiving software patents.

Don't be left behind! And remember, software can be simultaneously protected by a patent and a copyright. Indeed, until the patent is granted (typically, not until at least a year after the application is filed), the software can also be protected as a trade secret.

DDJ

You Know Kodak.

Or Do You?

Kodak digital science™

© 1995 Eastman Kodak Company

Historic developments are taking place at **Kodak**. The most powerful names in the communications and computer industries are forming key alliances with us to access our innovations in digital imaging. ■ Alliances with Microsoft, HEWLETT® PACKARD, SEGA®, Sprint®, IBM, and WANG are just the beginning. Explosive growth in the home office computer market and continued expansion of business networks around the globe have created an unparalleled demand for our digital imaging technology. ■ It has also created a demand for professionals who want to get in on the ground floor of these new, entrepreneurial enterprises.

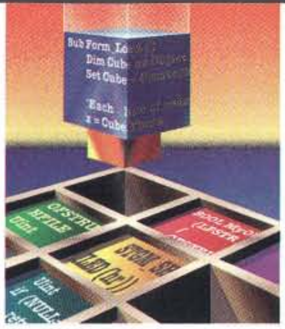
Software Engineers, Software Quality Engineers

Opportunities are available in the following product areas: Photo CD; Large Volume Optical Disk Storage Systems; PostScript Printer Software Development/RIPS; Photofinishing and Photographic Equipment.

These positions require a BS (or equivalent); 3-5 years product development experience with C/C++; Object Oriented; GUI (Motif); UNIX/Mac/MS Windows/Windows NT; Windows 95; OS2; Real Time Machine Control Applications; Test Engineering and SW Quality Processes; and Database Design/Development (FoxPro, Sybase).

In addition to a stimulating technical environment, you will enjoy our location in Rochester, New York. Affordable housing, high caliber schools, diverse cultural and recreational activities, and the beauty of the Finger Lakes Region all contribute to an enviable lifestyle.

Kodak offers competitive compensation, a liberal benefits package and relocation assistance. Qualified candidates are invited to forward resumes to: **Eastman Kodak Company, Attn: Professional Staffing, Dept. BDAIDDS36, 343 State Street, Rochester, NY 14650-1139. FAX: (716) 724-9416.** Eastman Kodak Company is an equal opportunity employer. We invite people with disabilities to notify us of their need for accommodation. NO PHONE CALLS, PLEASE.



Observations on Observer

Erich Gamma and Richard Helm

Partitioning a system into objects is a key activity during object-oriented design. As a result of this partitioning, we may create objects that depend on other objects. Changes in one object must be reflected into others. There are many different ways to ensure that these dependencies are maintained.

For example, consider the timer object in Figure 1, which keeps the current time, and a digital-display object that shows the current time. Whenever the timer ticks, this time-display object has to be updated. In other words, the time-display object has to maintain the constraint to always reflect the timer's current time. A simple solution is to connect the timer object directly to the time-display object. Whenever the timer changes, it explicitly tells the display object to update itself. Figure 2 shows the corresponding class diagram for directly coupling the timer with its observer. Listing One (listings begin on page 63) is one way to implement this in C++. While this direct coupling of two objects is simple to implement, it can also introduce problems in different areas:

- Reusability. It is not possible to reuse the timer independently of the time display. The two objects are strongly coupled and must always be used together, even when the client is only interested in the timer.
- Maintainability. The direct coupling makes maintenance more difficult. It is not possible to test or port the timer to a different platform independently of the time display.
- Extensibility. Whenever you want to add another kind of timer display (say, an analog display) that needs to be synchronized with the timer, you have to modify the *Timer* class to also update this new kind of time display.

These problems are clearly not serious in this simple example. However, in the context of a larger system we have to be more alert when introducing dependencies among objects. In fact, it is a key design activity to control and manage the dependencies between objects. Badly managed dependencies can result in a tangled system that is hard to reuse, maintain, or extend. A common theme of several patterns in our book *Design Patterns: Elements of Reusable Object-Oriented Software* is how to break hard dependencies by decoupling the involved objects. The Observer pattern is one of them.

The intent of the Observer pattern is to define dependency relationships between objects so that when one changes, its dependents are notified and can update themselves accordingly. The Observer pattern enables objects to observe and stay synchronized with another object without coupling the observed object with its observers. The pattern has two participants: 1. a subject, and 2. the subject's dependent observers. Each time the subject changes, it is responsible for notifying its observers that it changed. Observers must ensure that whenever they are notified, they in turn make themselves consistent with their subject. A subject needs an interface that allows observers to subscribe and register their interest in changes to the subject.

The subject usually maintains a list of subscribed observers. Figure 3 illustrates these class relationships in OMT notation. Notice that we introduced two new base classes. The *Subject* class defines the mechanism for registering and notifying observers and the *Observer* class defines the update interface. This diagram illustrates how the Observer pattern breaks the direct coupling between *Subject* and *Observer*. The *Subject* knows nothing about its *Observers* except that they can be sent *Update* requests. This is because the reference from *Subject* points to the abstract class *Observer*. For this reason, we refer to this kind of coupling as "abstract." The abstract coupling between *Subject* and *Observer* resolves the problems mentioned previously:

- Reusability. The timer object can now be reused and distributed without the time-display object. It only has to be bundled with the abstract Observer class. Figure 4 illustrates the resulting class coupling.
- Maintainability. The objects are no longer directly coupled, and the timer object can be tested independently of the time display. For example, when you port the *Timer* class to another platform, you can test it as soon as you've ported it and Observer. You no longer have to wait for the testing until the time display and its associated graphical infrastructure are ported as well.
- Extensibility. It is easy to add additional objects that need to be synchronized

Richard is a consultant with DMR Group, an international information-technology consulting firm. He can be reached at Richard.Helm@dmr.ca. Erich is a software engineer with Taligent Inc. He can be reached at Erich_ Gamma@Taligent.com. Erich and Richard are coauthors of the award-winning book Design Patterns: Elements of Reusable Object-Oriented Software *(Addison-Wesley, 1994).*

with the timer. For example, an analog time display only needs to inherit from Observer, implement the *Update* interface, and register itself with the timer.

Figure 4 shows that the concrete observer knows the class of the subject it is observing, and it can rely on this interface to query the subject's current state.

The *Subject* and *Observer* classes (Listings Two and Three, respectively) illustrate how you could implement *Observer* in the context of the timer example. The key point about *Timer* is that its *Tick* member function calls *Notify*, which will call *update* on all its *Observers*.

Listing Four presents the classes *Observer* and *DigitalTimeDisplay*. *DigitalTimeDisplay* maintains a reference to the timer. Whenever the *Timer* ticks, it calls *Notify*, which in turn calls *Update* on its attached *Observers*. In this case, *DigitalTimeDisplay* receives the *Update* request, reads the time from the timer, and displays the time.

Notice how the *Timer* has no knowledge of how it is displayed. In fact, you could add another timer, say an *AnalogTimeDisplay*, and it would also be updated whenever the *Timer* ticked.

Absorbing the *Subject* and *Observer* Classes

One possible simplification of the canonical observer-class structure is to absorb the *Subject* and *Observer* classes into existing classes. For example, the Microsoft Foundation Classes (MFC) use this kind of simplification. MFC supports multiple views observing a document (see "Adding Auxiliary Views for Windows Apps," by Robert Rosenberg, *Dr. Dobb's Sourcebook of Windows Programming*, March/April 1995). In MFC, the subject functionality is absorbed into *Document*, and *Observer* is absorbed into the *View* class. This solution is simpler since it requires fewer classes, but the dependency relationships can only be defined between instances of *Document* and *Views*.

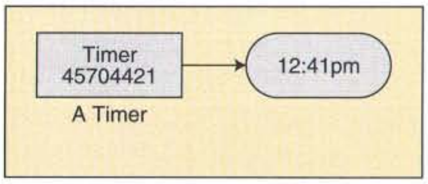


Figure 1: *Simple timer object.*

Inheritance Variations

There are also several variations in how inheritance is used to implement the Observer pattern. As Figure 2 shows, the interfaces for notifying and observing are defined by two classes. The *Observer* base class defines an interface consisting of an *Update* operation. In a language supporting multiple inheritance, *Observer* is often not a primary base class (that is, it is mixed in as an auxiliary base class). For example, in the timer example, *AnalogTimeDisplay* might need to inherit from a graphical base class like *View*. In this case, *AnalogTimeDisplay* mixes in the *Observer* class as an auxiliary class. Another variation is not to separate the notifying and observing interfaces into two separate classes. For example, in the Smalltalk-80 implementation of *Observer*, these two interfaces are supported by the universal *Object* class. Thus, each object in the system can act as both subject and observer. This is particularly convenient in a language that does not support multiple inheritance or in class libraries that don't want to rely on it. Using separate classes for subject and observer would require you to inherit from both subject and observer when an object needs to act as both.

In Figure 3, the *Timer* subclass inherits the *Subject* interface without any overriding. This is not always the case. For example, it is possible that a *Subject* subclass wants to customize how observers are maintained. In Smalltalk-80 the *Subject* base class (*Object*) implements the subject interface in a space-efficient way. Instead of storing the list of observers in each subject, the subject/observer mapping is maintained in a central dictionary. Only subjects that actually have observers are stored in the dictionary and have to pay for the subject service. However, this approach trades space for time: Accessing a subject's observers requires a dictionary look-up. For subjects that often notify observers, eliminate this inefficiency by storing the observers directly in an instance variable. The subject interface can then be implemented by accessing this list directly. In Smalltalk-80, this kind of *Subject* implementation is provided by the *Object* subclass *Model*. Consequently, the client has the choice between subject implementations with different tradeoffs by inheriting from either *Object* or *Model*. As an aside, the *Subject* interface is an example of so-called "coupled overrides." If you override one of the subject operations, you should also override the others.

Push versus Pull Update Protocols

In the timer example, the *Timer* makes no assumptions about what objects are observing it. Instead it relies on the various timer displays querying it to retrieve the current time. The observers "pull" the state of the subject to them. An alternative is for the timer to send, or "push," the time to its observers whenever it updates them. Pushing the time requires extending the interface of Observers to accept the time in seconds. To do this, you replace the *Observer* class with a *TimerObserver* class; see Listing Six. The *TimerSubject* class would now have to maintain a List of *TimerObservers* and its *Notify* function would look like Listing Seven. The observers are now more tightly coupled to the timer, but they no longer need to query the timer for the time. It is still possible to have arbitrary *Timer* observers by subclassing from *TimerObserver*. However, *TimerSubject* and *TimerObservers* can no longer be used to maintain general dependency relationships.

The decision to use push or pull update protocols depends on many tradeoffs: the amount of data being pushed and the expense of pushing it, the difficulty of determining what changed in the subject, the cost of notification and subsequent updates (whether subjects and observers are in the same address space), and dependencies introduced by observers being dependent on the pushed data.

The push model is more appropriate when editing text. Consider an implementation where a *TextSubject* stores the textual data and a *TextView* acting as its observer presents the text in window. When the user changes the text by entering a character, the pull model requires that the *TextView* completely reformat the text and refresh the window or that it somehow can determine which range of characters really changed. Both of these operations can be quite time consuming.

A more satisfactory approach is for the *TextSubject* to provide a "hint" of its changed text. The *TextView* uses this hint to update itself more efficiently. Hints can be simple, enumerated constants that provide general indications of what changed in the *Subject*, or more sophisticated, specific information to aid the *TextView*. *TextView* is interested in *how* the *TextSubject* changed—whether characters were added or removed, and where.

The hint can package information about the actual changes ("deleted range 12–27") and push it to the observers. The hint es-

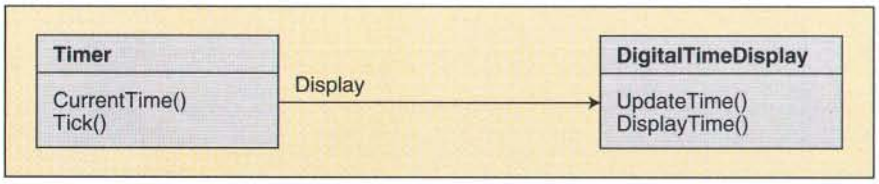


Figure 2: *Class diagram for directly coupling the timer with its observer.*

sentially sends the deltas that have occurred in the subject. In practice, not all observers will be interested in every hint; they may ignore some and act as if they had received a simple update request.

A hint can be extended with additional information by making it a first-class object. This enables subjects to bundle the additional information by subclassing from a *Hint* base class. At the receiving end, the observer downcasts the hint to the desired type and extracts the additional information. This downcast should of course not be done in a "hard" way, and it should by guarded by using the C++ run-time type identification facilities (*dynamic_cast*).

Who Sends Out Notifications

When notifications are sent, the subject must be in a consistent state. If it is not, strange results may occur in the observers as they try to update themselves from a nonsensical subject.

Which object has the responsibility to actually send the notification is also important. In our example, it is the *Timer* object that sends notifications in its *Tick* operation. This works fine as long as the subject is simple.

In Listing Eight, the *Tick* operation is overridden in a special kind of timer that allows you to set alarms. See the problem? When subjects have the responsibility to send notifications, overriding operations in the subject may cause spurious and inconsistent notifications. By overriding *Tick*, the first notification (sent from *Timer:: Tick*) is sent while the *AlarmedTimer* is in an inconsistent state (the *_alarm* variable should be set at that time but isn't until the second notification). Some observers could set off alarms by testing the result of *AlarmSet* and some could do so by testing the equality of *AlarmTime* and *CurrentTime*.

There are simple fixes for this problem. But for more complex subjects with derived classes, overriding operations that send notifications in the subject could make the subject inconsistent or cause duplicate notifications to be sent.

One solution to sending notifications to the subject is making clients change the subject to initiate the notifications. Whenever the client makes a change, it must call *Notify* on the subject. This solution is practical, but places extra burden on the clients. It is easy to forget to call *Notify* on the subject. Another solution is to define *Tick* as a template method (see the Template Method pattern from our book) that first just calls the operation *DoTick* and *Notify*. The *Timer* class defines *DoTick* to increment the current time. Subclasses can override this operation to provide their own extensions to the *Tick* operation.

Subscribing to Specific Aspects of Observers

When a subject has complex internal state, observers may spend much effort to determine exactly what changed in the subject. Along with using hints, you can reduce this burden by relying on intrinsic properties of the subject itself. Complex subjects may only change their state in predefined ways. Changes in part of a subject's state may be independent of changes in other parts.

Such properties can be exploited by having the subject define independent aspects and having observers only subscribe to the aspects they are interested in.

In the *Timer* example, suppose that the *Timer* class were implemented with three distinct counters that maintained the time in seconds, minutes, and hours. Now the various timer displays will usually be defined in terms of hours, minutes, and seconds. Clearly, not all of these need to be updated each second. In fact, the hour, minute, and second counters change almost independently of each other. You can exploit this by defining aspects that represent changes in hours, minutes, and seconds and defining our displays as consisting of three independent parts, each subscribing to a particular part of the *Timer*.

In this example, assume that the aspects are simply defined as integer constants that are passed as a parameter to *Notify*; see Listing Nine. The class *Timer* makes its aspects available to the clients as class-scoped constants. In Listing Ten, for example, the changed aspect is passed as a hint to the *Observer's Update* operation in Listing Eleven. If there are many different aspects, the update operation becomes a lengthy conditional statement that maps an aspect to a piece of code. Such conditional code is not very elegant. There are different techniques to avoid this kind of manual-dispatching code. One technique is demonstrated in VisualWorks Smalltalk, wherein the dispatching problem is solved with a *DependencyTransformer* object that implements the Observer interface. It knows which aspect it is interested in and keeps track of the actual receiver of the notification and the operation to be executed by the receiver when the aspect changes. Figure 5 shows a possible class structure for a *DependencyTransformer*.

When a *DependencyTransformer* receives the update, it checks the aspect. If the aspect matches, *DependencyTransformer* invokes the operation on the Receiver. *DependencyTransformers* are created by the subject when an observer expresses its interest in a changed aspect. This requires a way to specify the operation to be called. In Smalltalk, the operation's selector name is specified; *#updateSeconds*, for example.

DependencyTransformers act as an intermediary between the subject and its dependent object. They map the Observer interface to an operation of the dependent object. A *DependencyTransformer* is therefore an example of the *Adapter* pattern.

Making an Arbitrary Class a Subject

Sometimes classes are not designed to be subjects, but later, you realize that instances of these classes might have dependent objects. How do you make such classes into subjects? You could change the class by mixing in the *Subject* interface, but this is not always possible. The class you wish to make a subject may not be modifiable—it may reside in a class library over which you have no control.

An elegant way to allow arbitrary classes to become subjects is to wrap the object in another object that adds the *Subject* behaviors and interfaces. This decorator object (this is an example of the *Decorator* pattern) intercepts and forwards all requests to the wrapped object, and

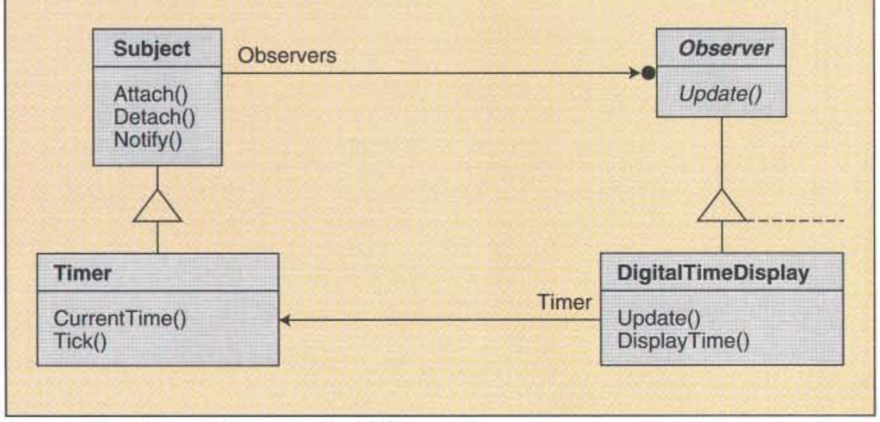


Figure 3: *Class relationships in OMT notation.*

©1994 EDF

Thanks to you, all sorts of everyday products are being made from the paper, plastic, metal and glass that you've been recycling.

But to keep recycling working to help protect the environment, you need to buy those products.

BUY RECYCLED.

AND SAVE.

So look for products made from recycled materials, and buy them. It would mean the world to all of us.

For a free brochure, write *Buy Recycled*, Environmental Defense Fund, 257 Park Ave. South, New York, NY 10010, or call 1-800-CALL-EDF.

ENVIRONMENTAL DEFENSE FUND EDF

Ad Council A Public Service of This Publication EPA

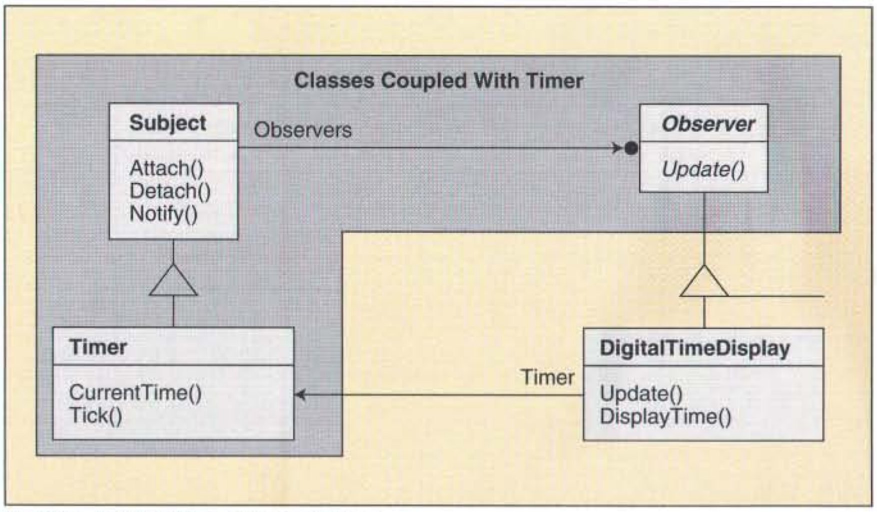


Figure 4: *Resulting class coupling.*

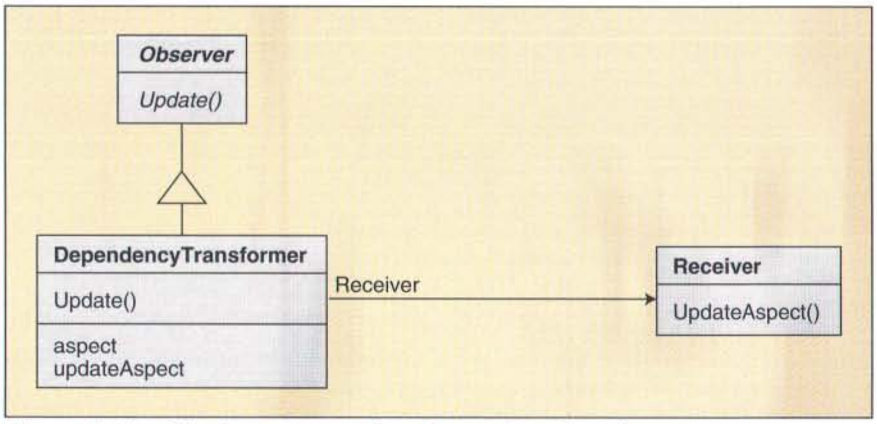


Figure 5: *Possible class structure for a* DependencyTransformer.

notifies clients after operations which are likely to change the wrapped object.

Suppose the class *Timer* was in fact not designed to be a subject and was defined as in Listing Twelve. You could make *Timer* a subject by defining a *TimerDecorator* class as in Listing Thirteen. The *TimerDecorator* has the same interface as the *Timer*, so it looks like a timer to clients. Every request made of the *TimerDecorator* is forwarded on to the timer, and then the timer calls *Notify* on itself to update its observers; see Listing Fourteen.

Making objects be subjects by using decorators is only practical when the decorated object's interface is relatively small, because you have to duplicate the subject's interface in the decorator. If the subject's interface is large, this approach can become unwieldy.

Conclusion

In one form or another, the Observer pattern occurs in many object-oriented systems. While most commonly used for decoupling user interfaces from data to be displayed on the user interface, often it is used to manage dependencies between objects. The Observer pattern has many more possible variations than the few we've examined. For example, we did not look at batching notifications, concurrency and distribution, or observing more than one subject.

Finally, a description of the Observer pattern would not be complete without mentioning its origin in the Smalltalk's Model-View-Controller (MVC) framework. In this design, the Model encapsulates application data. The View presents the model to the user. The controller is responsible for handling user input. From a dependency-management view, MVC provides the idea of decoupling the application data from the user interface. The benefit of this decoupling is that the application data can be presented by different user interfaces. In MVC terminology, the timer object becomes the "model" and the time display becomes a "view." If we supported manipulation of the time display by the user, then this behavior would be assigned to the *Controller*.

DDJ

Listing One *(Text begins on page 59)*

```
class DigitalTimeDisplay;
class Timer {
public:
   Timer(DigitalTimeDisplay*);
   long CurrentTime() const;
   void Tick();
private:
   DigitalTimeDisplay* _display;
   long _currentTime;
};

class DigitalTimeDisplay {
public:
   DigitalTimeDisplay();
   void DisplayTime();
   void UpdateTime(long time);
};

void Timer::Tick()
{
  _currentTime++;
  _display->UpdateTime(_currentTime);
}
```

Listing Two

```
class Subject {
public:
  void Attach(Observer*);
  void Detach(Observer*);
  void Notify();
protected:
  Subject();
private:
  List<Observer*> *_observers;
};

void Subject::Notify () {
  ListIterator<Observer*> i(_observers);
  for (i.First(); !i.IsDone(); i.Next() ) {
        i.CurrentItem()->Update();
  }

}
```

Listing Three

```
class Timer : public Subject {
public:
    Timer();
    virtual void Tick();
    long CurrentTime()
const;
private:
    long _currentTime;
}

void Timer::Tick()
{
   _currentTime++;
   Notify();
}
```

Listing Four

```
class Observer {
public:
   virtual void Update() = 0;
protected:
   Observer();
};

class DigitalTimeDisplay : public Observer {
public:
   DigitalTimeDisplay(Timer*);
   virtual void Update();
void DisplayTime(long time);
private:
   Timer* _timer;
};

DigitalTimeDisplay::DigitalTimeDisplay(Timer* t) : _timer(t)
{
}

void DigitalTimeDisplay::Update()
{
   DisplayTime( _timer->CurrentTime() );
}
```

Listing Five

```
class AnalogTimeDisplay : public Observer {
public:
   AnalogTimeDisplay(Timer*);
   virtual void Update();
   void DisplayTime(long time);
private:
   Timer* _timer;
};

AnalogTimeDisplay::AnalogTimeDisplay(Timer* t) : _timer(t)
{
}

void AnalogTimeDisplay::Update()
{
   DisplayTime( _timer->CurrentTime() );
}
```

Listing Six

```
class TimerObserver {
public:
   virtual void Update(long) = 0;
protected:
   TimerObserver();
};
```

Listing Seven

```
void TimerSubject::Notify (long time)
{
    ListIterator<TimerObserver*> i(_observers);
    for (i.First(); !i.IsDone(); i.Next() ) {
        i.CurrentItem()->Update(time);
    }

}
```

Listing Eight

```
class AlarmedTimer : public Timer {
public:
   AlarmedTimer();
   virtual void Tick();
   long AlarmTime();
   bool AlarmSet();
private:
   long _alarmTime;
   bool _alarm;
};

AlarmedTimer::AlarmedTimer()
   : _alarmTime(0), _alarm(false)
{
}

void AlarmedTimer::Tick()
{
   Timer::Tick();
   if ( CurrentTime() == _alarmTime ) {
       _alarm = true;
   } else {
     _alarm = false;
   }

}
```

TOOT YOUR

OWN HORN

with DDJ Reprints

Increase credibility, professionalism, and sales with beautiful four-color, reprints of your latest product review, article, or column from *Dr. Dobb's Journal.*

Use Reprints For:

- Media Kits
- Marketing /Public Relations
- Trade Shows
- Seminars /Workshops
- Sales Tools

Call Sara Wood at (415) 655-4193

(continued from page 63)

Listing Nine

```
class Subject {
   //...
   void Notify(int aspect);
  //...
};
```

Listing Ten

```
class Timer: public Subject {
public:
   //...
   static const int ASPECT_SECONDS;
   static const int ASPECT_MINUTES;
   static const int ASPECT_HOURS;
   //...
   int Seconds() const;
   int Minutes() const;
   int Hours() const;
private:
   int _seconds;
   int _minutes;
   int _hours;
};

void Timer::Tick()
{
   _seconds = ++_seconds % 60;
   Notify(ASPECT_SECONDS);
   if ( _seconds == 0 ) {
       _minutes = ++_minutes % 60;
       Notify(ASPECT_MINUTES);
   }

   if ( _seconds ==0 && _minutes == 0 ) {
      _hours = ++_hours % 24;
      Notify(ASPECT_HOURS);
   }

}
```

Listing Eleven

```
class AnalogTimeDisplay : public Observer {
public:
   AnalogTimeDisplay(Timer*);
   virtual void Update(int aspect);
   void DisplayTime(long time);
private:
   Timer* _timer;
};

void AnalogTimeDisplay::Update(int aspect)
{
   if (aspect == Timer::ASPECT_SECONDS)
        // update second hand ...
   else if (aspect == Timer::ASPECT_MINUTES)
       // update minute hand ...
   else if (aspect == Timer::ASPECT_HOURS)
      // update hour hand ...
   else
      // full update
}
```

Listing Twelve

```
class Timer {
public:
   virtual void Tick();
   long CurrentTime() const;
private:
   long _currentTime;
};

void Timer::Tick()
{
   _currentTime++;
}
```

Listing Thirteen

```
class TimerDecorator : public Timer, public Subject {
public:
   TimerDecorator(Timer*);
   virtual void Tick();
private:
   Timer* _timer;
};
```

Listing Fourteen

```
void TimerDecorator::Tick ()
{
  _timer->Tick();
  Notify();
}
```

End Listings

The Greatest Thing Since Sliced Bread

No matter how you slice it, our free Consumer Information Catalog is great food for thought.

The Catalog dishes up more than 200 free and low-cost government booklets on subjects you can really sink your teeth into — such as staying healthy, eating right, finding a job, getting federal benefits, using credit, buying a home, helping your kids, taking care of your car, and much, much more.

To get your free copy, send your name and address to:

Consumer Information Center
Department GT
Pueblo, Colorado 81009

A public service of this publication and the Consumer Information Center of the U.S. General Services Administration

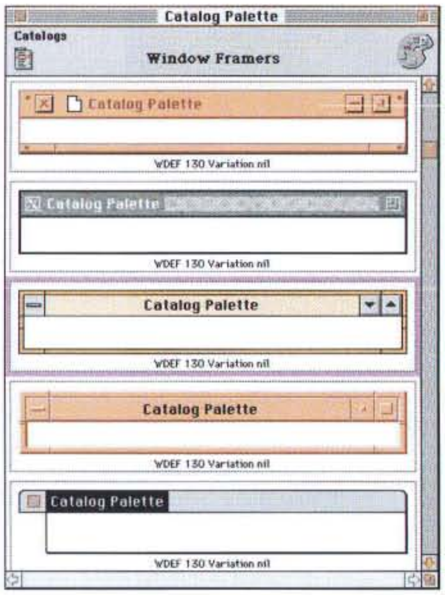
Building Internet And Other Native PowerPC Applications Has Never Been Easier Or Faster.

SmalltalkAgents®

Agents Object System
MACINTOSH

SmalltalkAgents (STA) is a sophisticated rapid application development environment based on a new generation of the Smalltalk language, enabling you to easily deliver double-clickable applications.

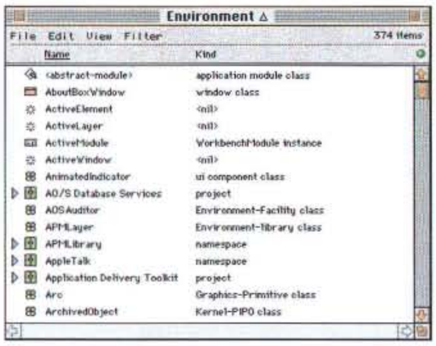
Copland Style GUI Look & Feel



Creating professional quality user interfaces is easy with our component parts libraries.

VisualWorkbench

Visually manipulate all objects including source and design elements using your mouse and keyboard. Visually manage



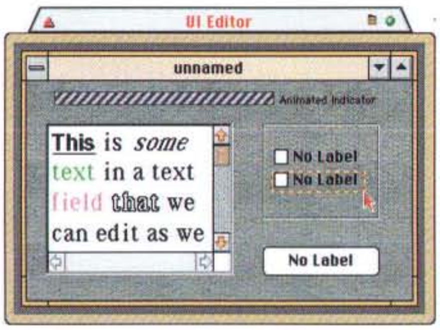
374 items

Name	Kind
<abstract-module>	application module class
AboutBoxWindow	window class
ActiveElement	<nil>
ActiveLayer	<nil>
ActiveModule	WorkbenchModule instance
ActiveWindow	<nil>
AnimatedIndicator	ui component class
AO/S Database Services	project
AOSAuditor	Environment-Facility class
APMLayer	Environment-library class
APMLibrary	namespace
AppleTalk	namespace
Application Delivery Toolkit	project
Arc	Graphics-Primitive class
ArchivedObject	Kernel-PIPO class

design and project elements in a "Finder-like" desktop workspace as fluidly as you work with folders and documents on your desktop. Interactively build, wire, and interconnect reusable components and interfaces in an integrated environment.

GUI Design & Generation

Live "Drag and Drop" manipulation to build your application's visual interface using components that "know" how to behave and autoconfigure themselves into an environment. Create new components and/or wire together existing components that can



be saved as reusable template designs for use in other applications or containers.

DTP Engine & Word Processor

STA not only includes a programmable word processor component and HyperMedia engine, but also a powerful report writer that supports embedding of any kind of objects, movies, flows, and international text, and page layout.

C/C++, Pascal Workbench

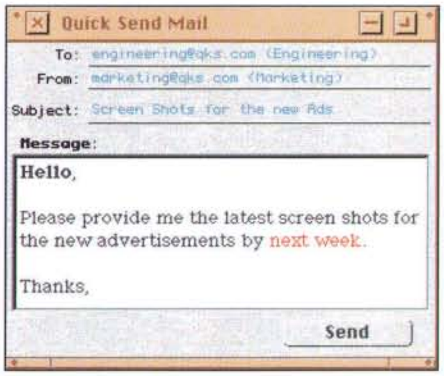
Compile, edit, and dynamically link C/C++, Pascal, Fortran, and Assembly code directly from within our STA VisualWorkbench as an integrated part of the Smalltalk application development process.

Component-based Architecture

STA components are designed for OpenDoc and OLE, and will give you transparent integration with OpenDoc and OLE when they become available.

Threading & Internet Tools

STA provides powerful support for Internet



To: engineering@qks.com (Engineering)
From: marketing@qks.com (Marketing)
Subject: Screen Shots for the new Ads
Message:
Hello,

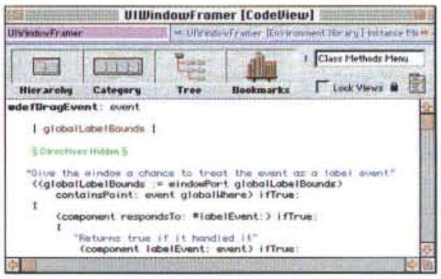
Please provide me the latest screen shots for the new advertisements by next week.

Thanks,

server as well as client tool development. Pre-emptive threading, thread safe libraries and classes for TCP/IP protocols are standard features enabling you to quickly and easily deliver custom e-mail, WWW, list-server, and other dial-up/network related apps.

PowerPC Support

STA provides binary portability across differ-


UIWindowFramer [CodeView]
UIWindowFramer
UIWindowFramer [Environment library] instance Me
Hierarchy
Category
Tree
Bookmarks
Class Methods Menu
Lock Views

```
edefDragEvent: event

   | globalLabelBounds |

   § Directives Hidden §

  "Give the window a chance to treat the event as a label event"
   ((globalLabelBounds := windowPort globalLabelBounds)
        containsPoint: event globalWhere) ifTrue:
   [
        (component respondsTo: #labelEvent:) ifTrue:
        [
           "Returns true if it handled it"
            (component labelEvent: event) ifTrue:
```

ent CPUs and Operating Systems. Design applications today on one platform and simply deploy on other platforms as required.

For further information, please contact Susan, Dept. D at 1-800-296-1339 or at <info@qks.com> or visit our Web site http://www.qks.com/.

QKS
Quasar Knowledge Systems, Inc.
9818 Parkwood Drive
Bethesda, MD 20814 USA
Tel:(301)530-4853 Fax:(301)530-5712

CIRCLE NO. 28 ON READER SERVICE CARD

MOTOROLA

Software Development Kits

Unlock the power of PowerPC™ microprocessors.

Motorola's Software Development Kits leverage our experience as one of the creators of the PowerPC Architecture™. We know how to get the highest possible performance out of PowerPC microprocessors. And we build that expertise into our highly optimizing compilers and tools for the Windows NT™, Mac OS™, AIX® and Solaris® operating systems.

Contact Motorola for more information and unlock the power of PowerPC microprocessors.

Call 1-800-347-8384 or 1-512-891-2999.

Internet: ppcinfo@risc.sps.mot.com http://www.mot.com/PowerPC/

CIRCLE NO. 28 ON READER SERVICE CARD